1994

A mobile robot sonar system with obstacle avoidance

Byrne, Patrick Gerard.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A MOBILE ROBOT SONAR SYSTEM WITH OBSTACLE AVOIDANCE

by

Patrick Gerard Byrne

March 1994

Thesis Advisor: Yutaka Kanayama

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1994	Master's Thesis

4. TITLE AND SUBTITLE
A Mobile Robot Sonar System With Obstacle Avoidance (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Byrne, Patrick Gerard

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The major problem addressed by this research is how to allow an autonomous vehicle to dynamically recognize changes in its environment, to map its environment, and alter its path to avoid obstacles while still reaching its goal point.

The approach taken was to modify existing sonar functions in previous work, to better utilize sonars, and to perform many experiments to determine what data to expect from sonars while the vehicle is in motion. By applying the linear square fitting algorithm, the robot has the ability to map the objects within sensor range of an autonomous vehicle.

The results are that, given an initial and goal point, the robot can proceed on a directed path, utilize its sonar sensor(s) used to detect obstacles, and when an obstacle is detected have the capability to dynamically compute a parallel path and smoothly alter its motion to the parallel path. The robot now has the capability to track the obstacle, and, once clear of the obstacle smoothly alter its motion to a path that will reach its goal point. The ability for the robot to combine smooth motion with obstacle avoidance has now been successfully programmed.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Autonomous vehicle, robot, obstacle avoidance, sonar sensing	102
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Approved for public release; distribution is unlimited

A MOBILE ROBOT SONAR SYSTEM WITH OBSTACLE AVOIDANCE

by

Patrick Gerard Byrne
Lieutenant, United States Navy
B.S., Bloomsburg University, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1994

Author: [redacted]

Patrick Gerard Byrne

Approved By: [redacted]

Yutaka Kanayama, Thesis Advisor

[redacted]

Man-Tak Shing, Second Reader

[redacted]

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The major problem addressed by this research is how to allow an autonomous vehicle to dynamically recognize changes in its environment, to map its environment, and alter its path to avoid obstacles while still reaching its goal point.

The approach taken was to modify existing sonar functions in previous work, to better utilize sonars, and to perform many experiments to determine what data to expect from sonars while the vehicle is in motion. By applying the linear square fitting algorithm, the robot has the ability to map the objects within sensor range of an autonomous vehicle.

The results are that, given an initial and goal point, the robot can proceed on a directed path, utilize its sonar sensor(s) used to detect obstacles, and when an obstacle is detected have the capability to dynamically compute a parallel path and smoothly alter its motion to the parallel path. The robot now has the capability to track the obstacle, and, once clear of the obstacle smoothly alter its motion to a path that will reach its goal point. The ability for the robot to combine smooth motion with obstacle avoidance has now been successfully programmed.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

Obstacle recognition is a common application of sonars used in many different applications. Motion control and path following for autonomous vehicles can be accomplished in a variety of ways. Whether or not the vehicle is in a static or changing world will drastically alter the path it will take. In previous research at the Naval Postgraduate School, obstacles detected by sonars were addressed by Solomon Sherfey in [Ref. 1] and resulted in an ability to use sonars that was not easily accomplished from a user standpoint. Sonars are used to determine location, and to recognize obstacles to be avoided. A high level language called MML (model bases mobile robot language) is the driving force behind the robot *Yamabico*. Real testing of software development and algorithms can be done on an autonomous vehicle. By restructuring the code and making it more modular, this will make MML more portable with the ability to apply its functions to other vehicles.

B. OVERVIEW

Although *Yamabico* may have precise knowledge of its location in a given environment, it is only capable of detecting the presence of unexpected obstacles in its path by relying on its 12 sonars that can be operated at anytime. They have an accuracy of approximately 1 centimeter, and are consistent in their results. *Yamabico* can move at a speed up to 65 centimeters per second in a translational motion, forwards or backwards. The sonars are low to the ground and will not pick up obstacles that are high, such as overhangs, or low obstacles below 38 centimeters. If an obstacle is detected, *Yamabico* has the ability to alter its path to avoid the obstacle, and once clear of the obstacle, return to its original path. Once the obstacle is recognized, *Yamabico* should smoothly transition to an alternative path and smoothly transition back to the original path once it can be done safely. Hence, the fusion of an obstacle being recognized and then having motion functions

available to smoothly deviate from the original planned path are the basic premises on which the sonar system for *Yamabico* has been designed and implemented.

II. PROBLEM STATEMENT

The problems we are addressing can be broken into two basic components.

1. The first is to construct a sonar function library, with a simple, transparent and efficient user interface.

2. The second component is to use these functions for real time obstacle avoidance. The sonar system can be used concurrently with the vehicle in motion, either translational or rotational. The sonars can be used to determine where an obstacle is, and for a vehicle to dynamically transition from its pre-planned path to another newly planned path in order to maintain the safest path while avoiding obstacles.

A. SONAR FUNCTION LIBRARY

One of the problems in past sonar functions has been that they were logically incorrect and unreliable. The C code was not written in ANSI C, so a lot of checking at compilation for such things as type and parameter matching being two examples, was never being done. A strong asset of ANSI C is that it's the next step towards C++, which then gives us the benefit of object oriented programming with inheritance.

The first step in improving sonar functions is to trace key algorithms to ensure that they are being properly implemented, and to ensure that all unnecessary code is removed. Over time, adjustments have been made to different functions, with needless variables and equations not being deleted. Later, more complex functionality, such as linear square fitting, was reorganized.

The second step in improving sonar functions is to make them more modular. That is to shorten the sonar code into smaller functions that are correct and easily understood. Making a module small to perform one basic operation for each is one step towards making the code more portable. By making the functions easily understood, the user should have no problem implementing the functions, and testing them on an actual platform.

B. OBSTACLE AVOIDANCE

Consider a vehicle traversing a hallway on a safest path trajectory, say the center of the hallway. Along the way the vehicle encounters an obstacle in its path. In order to continue on a safest path trajectory, the vehicle must change its linear path. This functionality is possible with the linear fitting and some other basic capabilities.

The introduction of obstacle avoidance as a function that can be called when planning the path of the vehicle is one way to give intelligence to the robot in navigating a path.

One simple way to do this is to (see Figure 1) detect the obstacle with a forward looking sonar, shift to a parallel path, and then resume its original path once the side detecting sonar detects no obstacle. This demonstrates the ability of a vehicle to dynamically alter its path for a safer path, and resume its original path once the obstacle has passed.

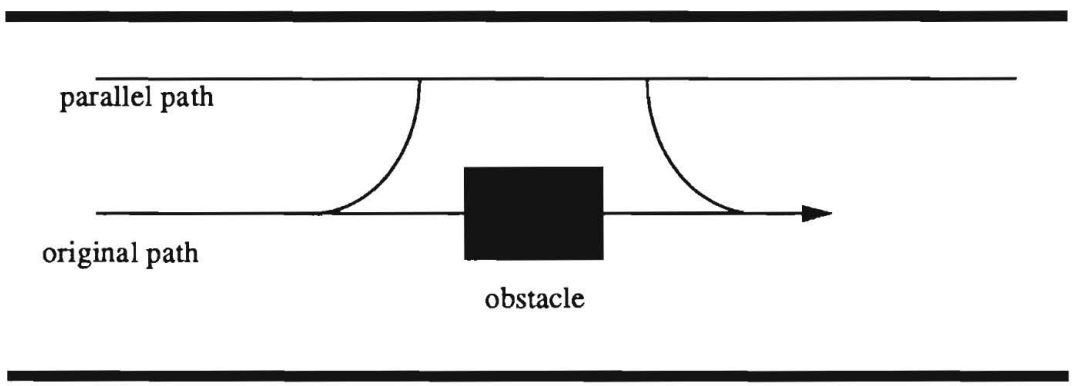

Figure 1: Example of obstacle avoidance

Writing functions that allow the vehicle to intelligently avoid obstacles and reach its goal as the world becomes more complex will be the approach taken. It is important to ensure each function is sound in solving one problem, and that different functions can be dynamically used as different situations arise.

As a path becomes more complex, and the number of obstacles increases, additional decision making becomes necessary. The challenge will be to cover as many different cases as possible, and to dynamically path plan in a timely manner.

III. SONAR HARDWARE SYSTEM

A. HARDWARE SYSTEM

Yamabico's sonar hardware is extremely efficient because a dedicated sonar board with a microprocessor controls the sonar sensors. Yamabico's main central processing unit is interrupted only when data becomes available from the sonar array. The sonar system provides user interface functions that control Yamabico's array of sonar range finders. At any point within a user's program, any of the 12 sonars may be enabled or disabled. This allows the user to operate a given sonar only when necessary for a particular application. When needed, the sonar system returns the latest reading of a specified sonar out of the twelve. This system design is far better than the primitive one in which a user must wait 30 milliseconds after he/she issues a command. A user's program can also be forced to "busy wait" until some sonar-based condition is satisfied. This feature is particularly valuable for obstacle avoidance. For example, a user's program could be written to wait until the forward looking sonar's range is less than distance *d*, then stop. A block diagram of the sonar system is provided in Figure 2.

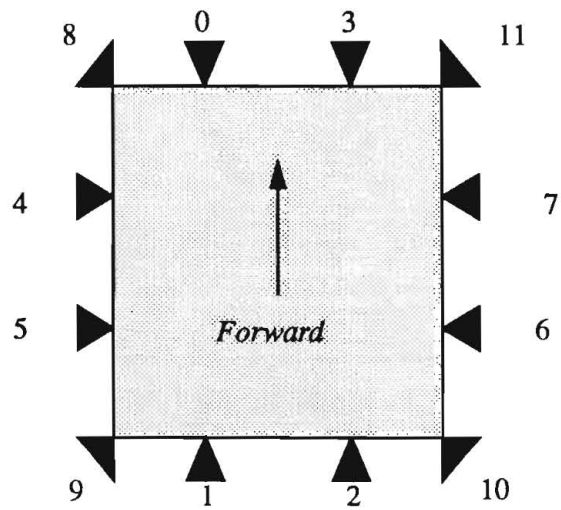

Figure 2: Yamabico sonars

Figure 3 shows the current hardware configuration of *Yamabico*.

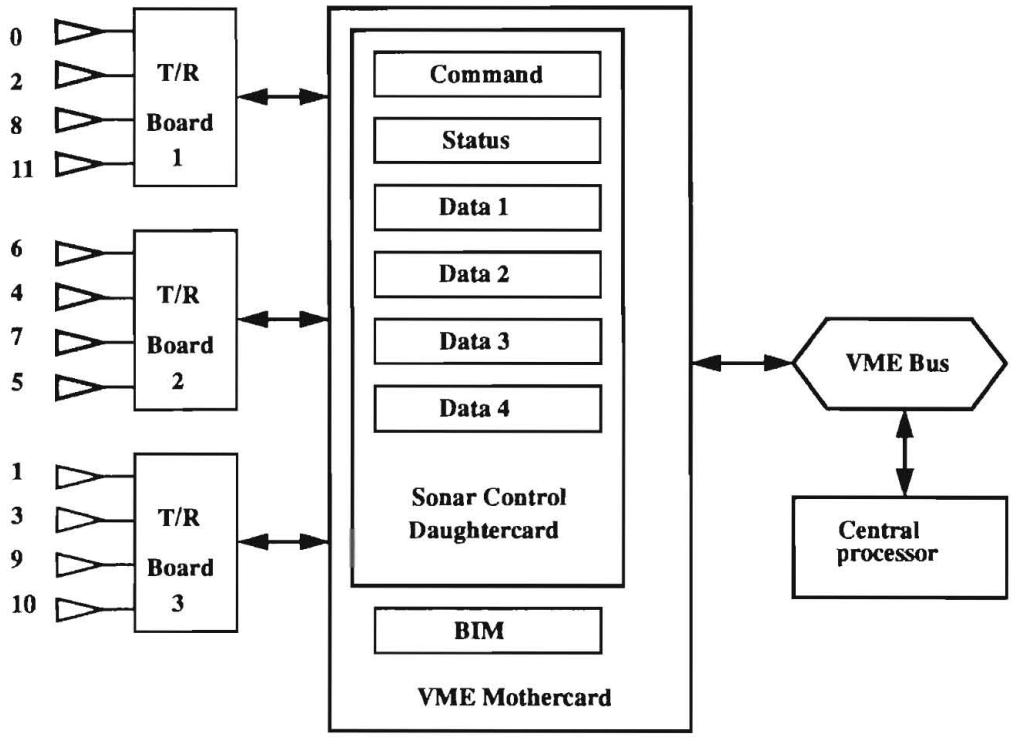

Figure 3: Sonar Hardware Architecture

1. Sonar Grouping

In order to reduce sampling time the sonars are operated in logical groups of four. The sonars of a logical group are all pulsed simultaneously and thus the sampling time is reduced by a factor of four as compared to individual firing of the sonars. The sonars of each logical group are oriented in such a way as to:

- prevent mutual interference
- provide a "look" in all four directions from each group
- present a similar aspect from each sonar during a rotational scan

Thus, logical group 0 consists of sonars 0, 2, 5 and 7 (see Figure 2), group 1 consists of sonars 1, 3, 4 and 6; group 2 consists of sonars 8, 9, 10 and 11; and group 3 is a

"virtual" group which consists of four permanent test values. The sonars of a group are symmetric about the robot's axis of rotation.

In addition to being *logically* grouped, the sonars are also *physically* grouped (see Figure 3). The physical grouping of the sonars is made to distribute the electrical load over the driver boards evenly and thus minimize any electrical transients associated with operation of the sonar. The physical grouping connects sonars 0, 2, 8 and 11 to driver/ amplifier board 1; sonars 4, 5, 6 and 7 to board 2; and sonars 1, 3, 9 and 10 to board 3. The reader will note that pairs of sonars from logical groups are assigned to physical groups, for example, sonars 0 and 2 from logical group 0 are assigned to physical group (driver/ amplifier board) 1.

Initial design of the control circuitry was based on two primary parameters: (1) a desired maximum range of 400 centimeter. and (2) a pulse width of 1 millisecond. Assuming a speed of sound in air, at sea level, of 340 meters/second we may calculate a round-trip time:

$$\text{round trip time} = \frac{400 \text{ cm}}{34000 \text{ cm/sec}} \times 2 = 23.53 \text{ msec} \qquad \text{(Eq 3.1)}$$

This round trip time is the period during which a valid echo may be received and is referred to as the *receive gate*. This interval is rounded up to 24 milliseconds and is derived by division of the sonar system's 2 MHz clock to ensure that the receiver is not falsely triggered by a direct path reception from it's adjacent transmitter, we opt to disable the receiver until the transmit pulse is complete. This will have the disadvantage of setting a minimum range equal to half the distance sound would travel in the time of a transmit pulse.

$$\text{minimum range} = 34000 \text{ cm./sec.} \times 1 \text{ msec.} \times 0.5 = 17 \text{ cm.} \qquad \text{(Eq 3.2)}$$

This minimum range lies approximately 9 centimeters outside the periphery of the robot. In order to allow the measurement of objects up to the periphery of the robot, the pulse width was decreased to 0.5 milliseconds thus reducing the minimum range to 8.5 centimeters.

In actual practice, the minimum range is set by firmware to 9.6 centimeters, the additional distance being due to time allotted for switching and settling in the circuitry.

All sonars of a logical group are pulsed simultaneously. Which groups are fired is determined by the value of the corresponding bit in the *command register* of the sonar control board, which in turn is set by the user with an MML function (see Figure 3). Hence, if bit 2 is set to 1 then group 2 sonars will be pulsed. If more than one group is selected to be pulsed, the sonar control board will pulse the first group on the list, and when the data from that pulse has been read from the fourth data register the sonar control board will proceed to the next group and pulse it, and so on in round robin fashion. Groups with their control bit set to 0 will *not* be pulsed. The sampling rate can thus be as high as 41 Hz with only one group enabled (based on a 24 millisecond read gate as determined in Equation 3.2) and will be halved for each additional group enabled. At a nominal robot speed of 30 centimeters per second. this sampling rate could provide an updated range within 0.75 centimeter of travel, exceeding our desired positional accuracy of 1 centimeter. Of course, real performance will be affected by any delay in reading the data registers due to other demands on the central processor (processing the sonar data, controlling motion, etc.).

2. Range Finding

There are four 16 bit data registers on the sonar control board, one for each of the four sonars in a logical group. When the transmit pulse is sent to the driver/amplifier boards a counter is started which increments each of the data registers every 6 microseconds. This time period is equivalent to a range of 1.02 millimeter:

$$\text{range} = 340000 \text{ mm/sec} \times 6 \text{ microsec} \times 0.5 = 1.02 \text{ mm} \qquad \text{(Eq 3.3)}$$

The incrementing of a particular data register continues until an echo is received or the range gate times out. The first 12 bits of the data register are allotted for range accumulation, thus allowing for a maximum range of 4.177 meters (4095 x 1.02 mm.). If the range gate should time out before an echo is received, the high bit of the over ranged sonar's data register is set to 1. This is the "over range" bit and is used to signal the ensuing

software that no echo was received. Bits 12, 13 and 14 of the data registers are not used. When the ranging cycle is complete, the appropriate group number is written into bits 4 and 5 of the status register and the "ready" bit, bit 7 of the status register, is set to 1. The ready bit is used as a flag when operating in the polled mode; i.e. without interrupts.

3. Interrupt Control

The sonar control board is actually a daughtercard which rides on a VME bus mothercard. The mothercard carries address decoders, bus drivers and interrupt control circuitry in the Bus Interface Module (BIM).

When the sonar has completed a ranging cycle an interrupt request is provided to the BIM. The BIM's *control register* holds information which determines whether an interrupt is to be generated or not, and if so which interrupt level is to be generated. Presuming an interrupt is generated, when the correct acknowledgment returns on the address lines the BIM's *vector register* provides the vector table entry where the central processor may find the vector to the interrupt handler. The correct interrupt level, the interrupt enable bit and interrupt vector are loaded to the BIM during software initialization.

4. Data Transfer

Each of the data registers is individually addressed on the VME bus by a VME short address, as is the status register. Transferral of the data is extremely straightforward. The interrupt handler simply reads the correct register, masks out the unwanted bits and writes the data to the stack. When the last data register is read, the sonar system resets the data registers and commences a ranging cycle on the next sonar group in it's round robin. The system will continue to operate autonomously until all the sonars are disabled.

IV. BASIC SONAR FUNCTIONS

A. DISTANCE

There are two functions available to return sonar values. One function, sonar() will return the range from the sonar to the object it is getting the return from. If there is no return, then a value of infinity is assigned, and for *Yamabico* this value is 999999. The infinity value is used for trouble shooting purposes, to detect whether or not there are instances of no return from objects at a distance of less than 4 meters. The second range function available is global(), and this will return the x,y coordinates of where the return was detected in the world that the vehicle is in. This is useful in the vehicle making a map of its world with obstacles in it. These functions can be found in Appendix A.

B. GLOBAL POSITION CALCULATIONS

By utilizing the compose function described by [Ref. 3] and seen in Figure 4, we can determine the actual point in a 2D coordinate system. Let the following equations represent q_1 an q_2,

$$q_1 \equiv (x_1, y_1, \theta_1)^T \tag{Eq 4.1}$$

$$q_2 \equiv (x_2, y_2, \theta_2)^T \tag{Eq 4.2}$$

The composition of these transformations is defined as

$$q_1 \, o \, q_2 \equiv \begin{pmatrix} x_1 + x_2\cos\theta_1 - y_2\sin\theta_1 \\ y_1 + x_2\sin\theta_1 + y_2\cos\theta_1 \\ \theta_1 + \theta_2 \end{pmatrix} \tag{Eq 4.3}$$

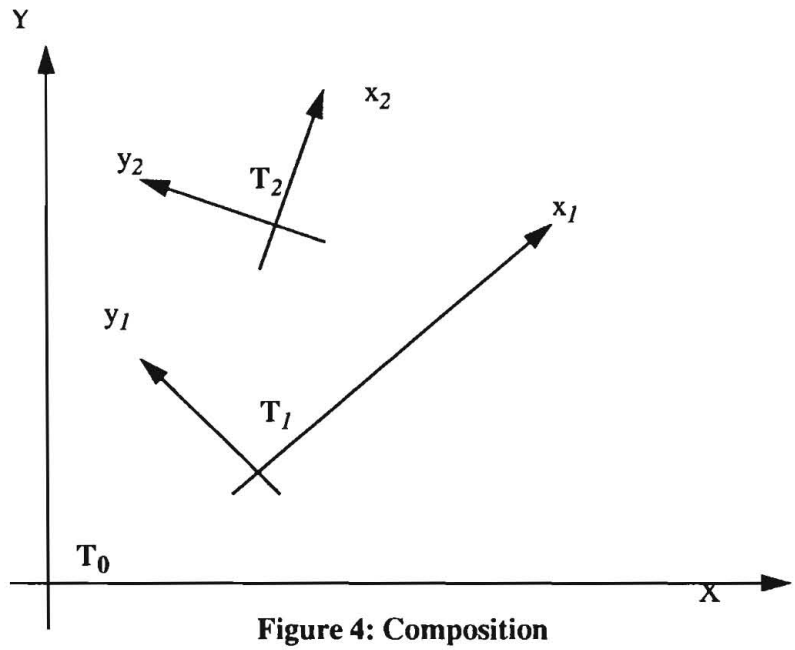

Figure 4: Composition

This functionality is extremely usefully in dynamically configuring new paths from your original paths. You can dynamically define another path depending on your position and the direction of your vehicle. For the sonar functions, it allows much more modularity to the code. The code is reusable, since the only thing unique to *Yamabico* are the actual sonar positions on the robot.

V. LINEAR FEATURE EXTRACTION

In addition to simple *range* and *point position* data, the sonar system recognizes the *linear features* of an orthogonal world. To do so we must provide some method for recognizing sets of data points which form the linear feature and a method for finding and describing the line segment that best fits that set of data points. This is accomplished in reverse fashion, i.e. we presume the data we are receiving belongs to such a set and continuously modify a descriptive line segment to a best fit of the data using a least squares fitting algorithm. This line segment continues to grow until the incoming data or certain measures of the line segment indicate that the line segment should be ended and a new one started. We use an implementation of least squares fitting described by [Ref. 1].

A. LEAST SQUARES FITTING

Suppose we have collected n consecutive valid data points in a local coordinate system, $(p_1,..., p_n)$, where $p_i = (x_i, y_i)$ for $i = 1,...,n$. We obtain the moments m_{jk} of the set of points

$$m_{jk} = \sum_{i=1}^{n} x^j{}_i y^k{}_i \qquad (0 \le j, k \le 2, \text{ and } j + k \le 2) \tag{Eq 5.1}$$

Notice that $m_{00} = n$. The *centroid* C is given by

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}\right) \equiv (\mu_x, \mu_y) \tag{Eq 5.2}$$

The secondary moments around the centroid are given by

$$M_{20} \equiv \sum_{i=1}^{n} (x_i - \mu_x)^2 = m_{20} - \frac{(m_{10})^2}{m_{00}} \tag{Eq 5.3}$$

$$M_{11} \equiv \sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y) = m_{11} - \left(\frac{m_{10} m_{01}}{m_{00}}\right) \tag{Eq 5.4}$$

$$M_{02} \equiv \sum_{i=1}^{n} (y_i - \mu_y)^2 = m_{02} - \frac{(m_{01})^2}{m_{00}} \qquad \text{(Eq 5.5)}$$

We adopt the parametric representation (r,α) of a line with constants r and α. If a point $p = (x,y)$ satisfies an equation

$$r = x\cos\alpha + y\sin\alpha \qquad (-\pi/2 < \alpha \leq \pi/2) \qquad \text{(Eq 5.6)}$$

then the point p is on a line L whose normal has an orientation α and whose distance from the origin is r (Figure 5). This method has an advantage in expressing lines that are perpendicular to the X axis. The point-slope method, where $y = mx + b$, is incapable of representing such a case ($m = \infty$, b is undefined).

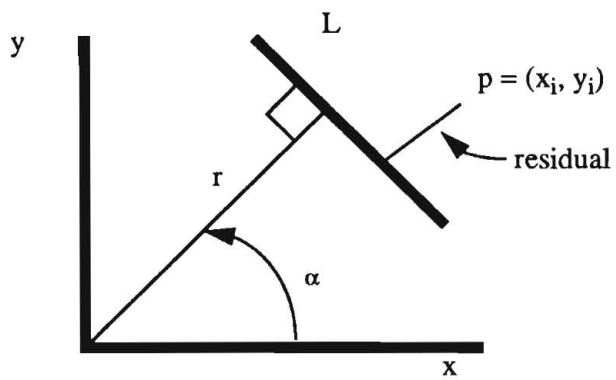

Figure 5: Representation of a line *L* using *r* and α

The residual of point $p_i = (x_i, y_i)$ and the line L = (r,α) is $x_i\cos\alpha + y_i\sin\alpha - r$. Therefore, the sum of the squares of all residuals is

$$S = \sum_{i=1}^{n} (r - x_i\cos\alpha - y_i\sin\alpha)^2 \qquad \text{(Eq 5.7)}$$

The line which best fits the set of points is supposed to minimize S. Thus the optimum line (r,α) must satisfy

$$\frac{dS}{dr} = \frac{dS}{d\alpha} = 0 \qquad \text{(Eq 5.8)}$$

Thus,

$$\frac{dS}{dr} = 2\sum_{i=1}^{n} (r - x_i\cos\alpha - y_i\sin\alpha) \qquad \text{(Eq 5.9)}$$

$$= 2\left(r\sum_{i=1}^{n} 1 - \left(\sum_{i=1}^{n} x_i\right)\cos\alpha - \left(\sum_{i=1}^{n} y_i\right)\sin\alpha \right) \qquad \text{(Eq 5.10)}$$

$$= 2\,(r m_{00} - m_{10}\cos\alpha - m_{01}\sin\alpha) \qquad \text{(Eq 5.11)}$$

$$= 0$$

and

$$r = \frac{m_{10}}{m_{00}}\cos\alpha + \frac{m_{01}}{m_{00}}\sin\alpha = \mu_x\cos\alpha + \mu_y\sin\alpha \qquad \text{(Eq 5.12)}$$

where r may be negative. Substituting r in Equation (5.7) by Equation (5.12) ,

$$S = \sum_{i=1}^{n} (\,(x_i - \mu_x)\cos\alpha + (y_i - \mu_y)\sin\alpha)^2 \qquad \text{(Eq 5.13)}$$

Finally,

$$\frac{dS}{d\alpha} = 2\sum_{i=1}^{n} (\,(x_i - \mu_x)\cos\alpha + (y_i - \mu_y)\sin\alpha)\,(-(x_i - \mu_x)\sin\alpha + (y_i - \mu_y)\cos\alpha) \qquad \text{(Eq 5.14)}$$

$$= 2\sum_{i=1}^{n}\left((y_i - \mu_y)^2 - (x_i - \mu_x)^2 \right)\sin\alpha\cos\alpha + 2\sum_{i=1}^{n} (x_i - \mu_x)\,(y_i - \mu_y)\,(\cos^2\alpha - \sin^2\alpha) \qquad \text{(Eq 5.15)}$$

$$= (M_{02} - M_{20})\sin 2\alpha + 2M_{11}\cos 2\alpha \qquad \text{(Eq 5.16)}$$

$$= 0$$

Therefore

$$\alpha = \frac{\operatorname{atan}\,(2M_{11} / (M_{02} - M_{20}))}{2} \qquad \text{(Eq 5.17)}$$

Equation (5.12) and Equation (5.17) are the solutions for the line parameters generated by a least squares fit.

B. FINDING ENDPOINTS

The residual of a point $p_i = (x_i, y_i)$ is

$$\delta_i = (\mu_x - x_i)\cos\alpha + (\mu_y - y_i)\sin\alpha \qquad \text{(Eq 5.18)}$$

Therefore, the projection, p'_i of the point p_i onto the major axis is

$$p'_i = (x_i + \delta_i\cos\alpha, y_i + \delta_i\sin\alpha) \qquad \text{(Eq 5.19)}$$

We will use p'_1 and p'_n as estimates of the endpoints of the line segment L obtained from the set p of data points.

C. RESIDUAL TESTING

We wish to do some pre-filtering of the data in order to remove points from the data stream which are clearly *not* colinear with the existing points of set p. In this way we can often detect the end of a line segment before having to perform the considerable computations necessary to include it in the line. If the point satisfies

$$\delta_{i+1} < \max(\sigma \times C1, C2) \qquad \text{(Eq 5.20)}$$

where C1 and C2 are positive constants (typically, C1 = 0.02 and C2 = 5.0) then the point can be included in the current line segment. C2 at 5.0 allows for more residual at a distance greater than 250 centimeters, up to 8 centimeters at a distance of 4 meters.

D. BEGINNING LINE SEGMENTS

First, the sonar returns must fall within their physical constraints. For Yamabico, acceptable return values fall between 9.3 centimeters and 409 centimeters. If a sonar return is not within this range, a segment will be generated if there have been at least 10 previous returns that met all requirements of the least square fitting to qualify as a segment.

Secondly, if it is the first return, you simply store it as the starting point and proceed with the next return.

With the line segment established, collection and testing of the additional data points can proceed. If the data point *passes* the residual testing, the moments and test values for the line are calculated including the new point. Should that test pass, the line segment

parameters (endpoints, length, etc.) are updated and the system proceeds to gather a new data point.

E. ENDING LINE SEGMENTS

There are three ways in which a line segment is ended. It may be ended by the failure of data points to pass the residual testing, explicitly ended by the sonar being disabled, or by the sonar return being outside the acceptable range.

VI. MML USER INTERFACE

A. GLOBAL CALCULATION

The compose function is implemented in a sonar function called calculate_global. It applies the compose function twice. The first time the compose function is used to determine the actual position of the sonar in the world being navigated by the vehicle, as seen in Figure 6. In this example Yamabico is at coordinates (80,40), in the "world coordinates". The sonars position on the robot is (9.5, -19.75). By applying the compose

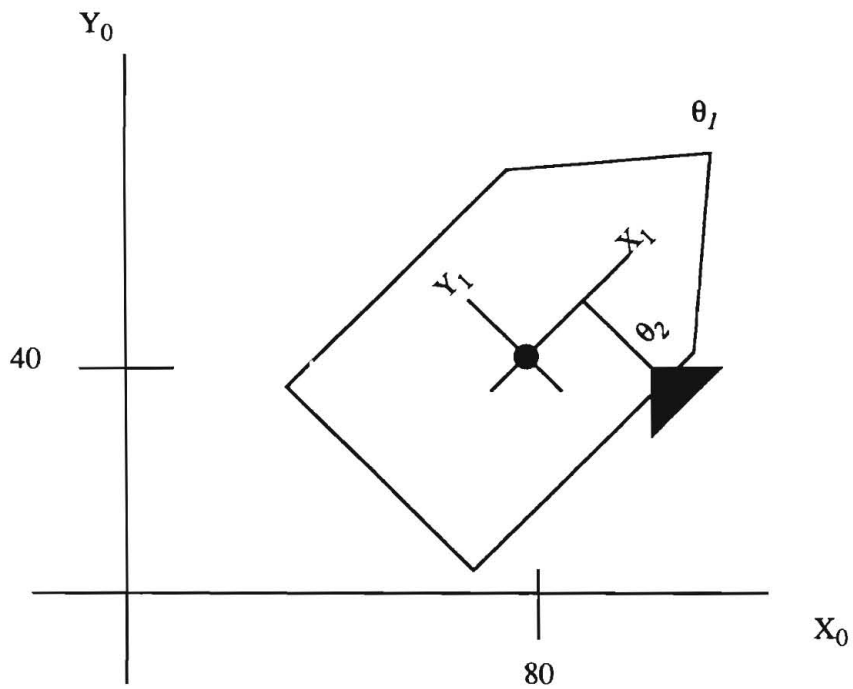

Figure 6: Example of first compose

function,

$$q_1 \circ q_2 \equiv \begin{pmatrix} x_1 + x_2\cos\theta_1 - y_2\sin\theta_1 \\ y_1 + x_2\sin\theta_1 + y_2\cos\theta_1 \\ \theta_1 + \theta_2 \end{pmatrix}$$

we determine the position of the sonar in "world coordinates". In this case it would be:

world sonar x coordinate = 100.68 = 80 + 9.5*cos(π/4) - (-19.75*sin(π/4))

world sonar y coordinate = 32.74 = 40 + 9.5*sin(π/4) + (-19.75*cos(π/4))

world sonar theta = $-\pi/4 = \pi/4 + -(\pi/2)$.

The second time compose is applied it determines where the sonar return is in the world being navigated by the robot, as in Figure 7.

In this case we apply the compose function and the results are:

sonar x coordinate from robot = 171.42 = 100.68 + 35*cos(-π/4) - 0*sin(-π/4)

sonar y coordinate from robot = -37.94 = 32.74 + 35*sin(-π/4) + 0*cos(-π/4)

which gives us the point in Figure 8.

By knowing where each sonar is on the vehicle (see Figure 9 and Table 1) and knowing where the vehicles position is, we can consistently determine where the object being detected is in relation to the world that Yamabico is in. This is needed so that a vehicle can dynamically map out the world.

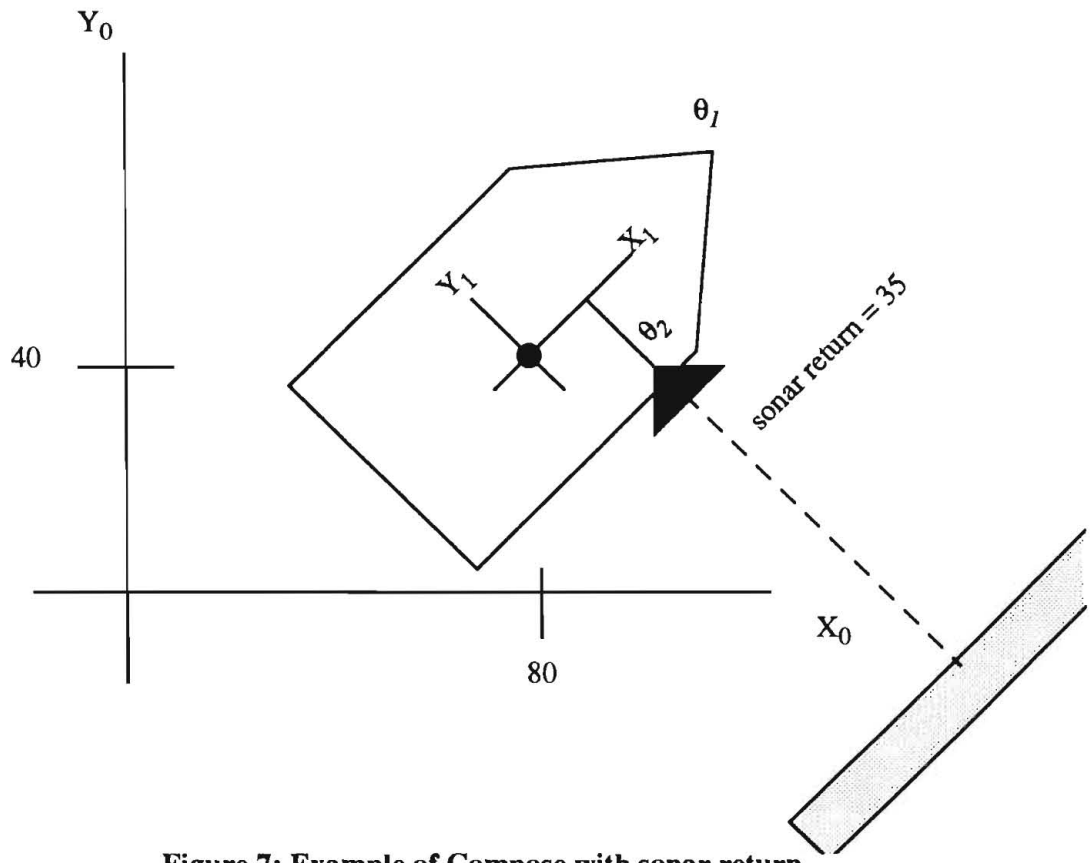

Figure 7: Example of Compose with sonar return

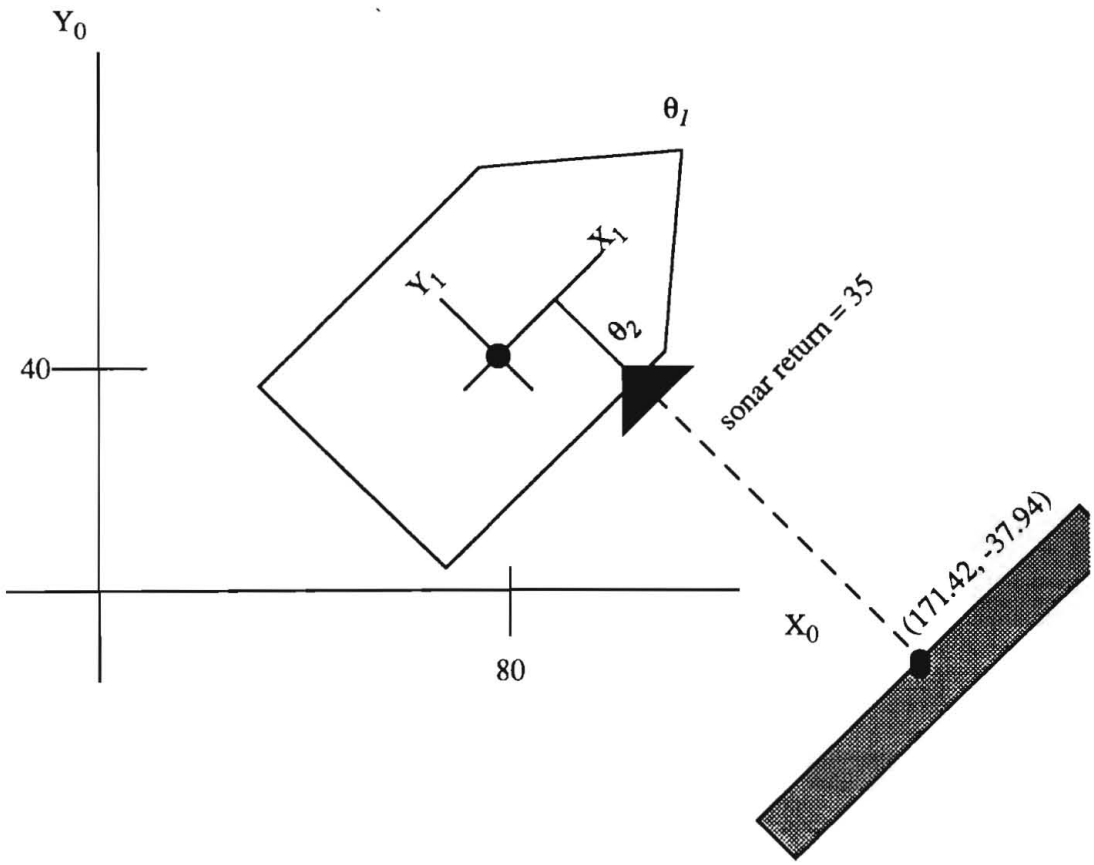

Figure 8: Result of compose on example

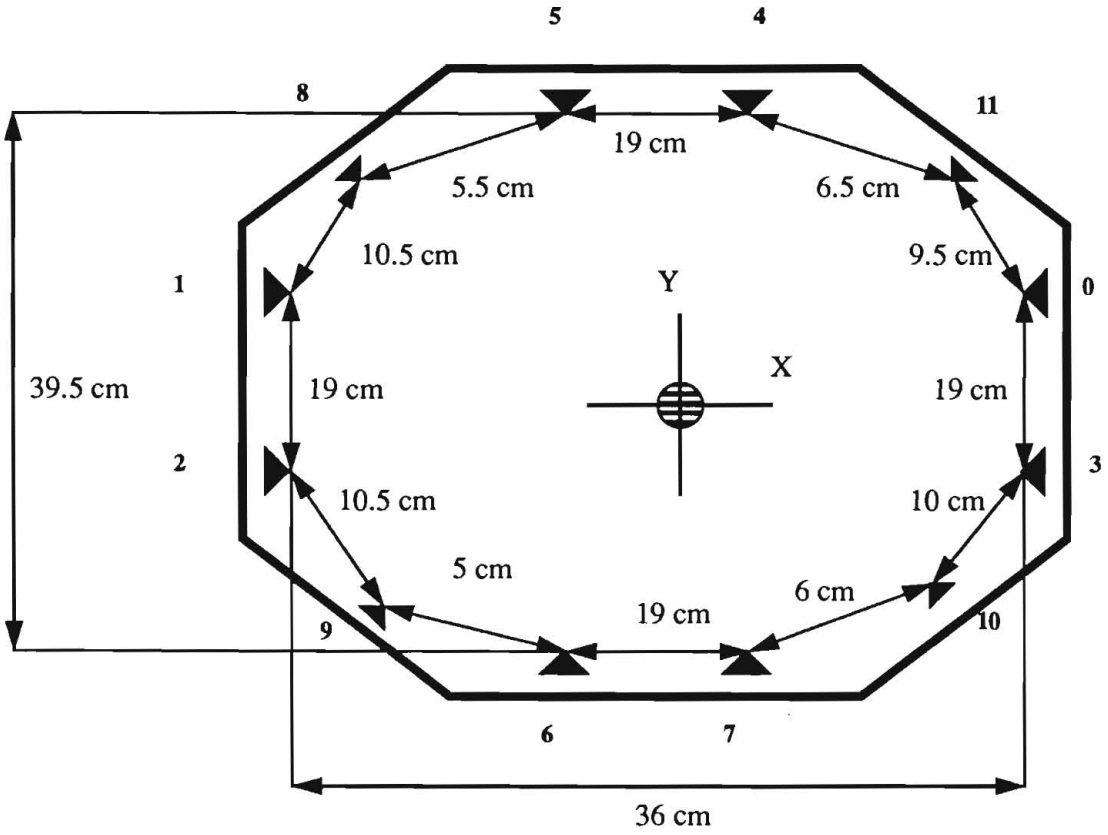

Figure 9: Sonar positions on Yamabico

Table 1: SONAR POSITIONS

Sonar	x	y	θ
0	18 cm	9.5 cm	0.0
1	- 18 cm	9.5 cm	π
2	- 18 cm	-9.5 cm	π
3	18 cm	-9.5 cm	0.0
4	9.5 cm	19.75 cm	π/2
5	-9.5 cm	19.75 cm	π/2
6	-9.5 cm	-19.75 cm	-π/2
7	9.5 cm	-19.75 cm	-π/2
8	-15 cm	19.75 cm	3/4 π
9	-14.5 cm	-19.75 cm	-3/4 π
10	15.5 cm	-19.75 cm	-π/4
11	16 cm	19.75 cm	π/4

B. SONAR FUNCTIONS

Sonar functions are found in sonarcard.c, sonarmath.c, sonario.c, sonarsys.c, and sonarlog.c, which are part of Yamabico's MML (model based mobile robot language), the name for the entire set of code for Yamabcio. The following functions and all sonar code can be found in Appendix A. The following are those functions which are available for use in the user.c and a brief description.

1. Enable Sonar

Syntax: void enable_linear_fitting(n)

int n;

Description:

The user calls this function passing in the sonar that is to be enabled. On Yamabico there are 12 available sonars. Each sonar should be enabled individually.

2. Disable Sonar

Syntax: void disable_sonar(n)

int n;

Description:

The user calls this function passing in the sonar that is to be disabled. On Yamabico there are 12 available sonars. Each sonar should be enabled individually.

3. Get Sonar Returns

Syntax: double sonar(n)

int n;

Description:

The user calls this function and passes in the sonar number that range data is wanted from. If no echo is received, then an INFINITY(1.0e6) is returned. If the distance is less than 10 cm, then a 0 is returned. If the sonar return is between 9 cm to 409 cm, then that floating point number will be returned in centimeters.

4. Get Global Sonar Returns

Syntax: posit global(n)

int n;

Description:

The user calls this function and passes in the sonar number that global range data is wanted from. The function will return a structure of type posit, which contains gx and gy, the global x and y coordinates.

5. Enable Linear Fitting

Syntax: void enable_linear_fitting(n)

int n;

Description:

The user calls this function and passes in the sonar number, so that linear fitting is applied to sonar returns. This will enable the robot to determine whether sonar returns are walls, or some type of linear surface.

6. Disable Linear Fitting

Syntax: void disable_linear_fitting(n)

int n;

Description:

The user calls this function and passes in the sonar that linear fitting is to be disabled on.

7. Set Parameters In Linear Square Fitting

Syntax: void set_sonar_parameters(c1, c2)

float c1,c2;

Description:

Allows the user to adjust constants which control the linear fitting algorithm. C1 is a multiplier to allow more leniency for greater sonar ranges, and C2 will adjust the tolerance allowed for sonar ranges being off the linear line being collected. Both are used to determine if an individual data point is usable for the algorithm. The default values are initialized to 0.02 and 5.0 respectively. For more information on C1 and C2 refer to Chapter V.C of this thesis.

8. Enable Data Logging

Syntax:

void enable_data_logging(n,filetype,filenumber)

int n,filetype,filenumber;

Description:

The user calls this function and passes in the sonar, the type of file data to be collected, and which file array (0, 1, 2, or 3) to collect the data in. There are three types of file data that can be collected. The first is raw data, the second is global data, and the third is segment data.

9. Disable Data Logging

Syntax: void disable_data_logging(n,filetype)

int n, filetype

Description:

The user calls this function and passes in the sonar, the type of file data to collected, and which file array (0, 1, 2, or 3).The type of file data that is to cease being collected is designated, either raw data, global data, or segment data.

10. Set Logging Interval

Syntax: void set_log_interval(n,d)

int n, d;

Description:

The user calls this function passing an integer designating how often the sonar data being collected should be written to the file collecting the data. The default value is 13, which for a speed of 30 centimeters per second and sonar sampling time of 25 milliseconds. would record a data point approximately every 10 cm. To collect all sonar data you pass in 1, so that every sonar return is recorded.

11. Transfer Raw Data To Host

Syntax: void xfer_raw_to_host(filenumber,filename)

int filenumber, filename;

Description:

The user calls this function and passes in the file number (0, 1, 2, or 3) and the name of the file that is to be created at the workstation to contain the raw sonar data collected.

12. Transfer Global Data To Host

Syntax: void xfer_global_to_host(filenumber,filename)

int filenumber, filename;

Description:

The user calls this function and passes in the file number (0, 1, 2, or 3) and the name of the file that is to be created at the workstation to contain the global sonar data collected.

13. Transfer Segment Data To Host

Syntax: void xfer_segment_to_host(filenumber, filename)

int filenumber, filename;

Description:

The user calls this function and passes in the file number (0, 1, 2, or 3) and the name of the file that is to be created at the workstation to contain the segment sonar data collected.

C. DATA LOGGING PROCEDURE

After *Yamabico* has completed its mission, recorded sonar data can be downloaded and checked to ensure that the hardware is performing optimally. The data that can be logged includes global sonar data, raw sonar data, segment sonar data, and the motion trace data of the robot. Once the robot has stopped, the data designated to be logged in user.c can now be downloaded. A message on the powerbook will instruct the user to connect the phone cable to the robot. Once the phone line is connected, the user must hit the space bar, then the character g, and the space bar once more. The data will then be downloaded to the

workstation. Once the download is completed, a bell sound will be heard from the powerbook on the robot. This is required for each type of data being logged.

VII. SONAR CHARACTERISTICS EXPERIMENTAL RESULTS

To be able to successfully use sonars in path navigation and obstacle avoidance, it is necessary to understand what data you can expect in different situations using sonars. This way you can determine in which cases you will be able to successfully avoid obstacles, and in which cases you will be unable to determine a safe path with only input from the sonars.

A. CASE 1

The robot is moving using its right sonar in a translational scan as in Figure 10. You would expect to get very accurate data and to be able to recognize the wall. As Figure 11 and Figure 12 show, this is the case. As expected, the robot can determine that there is a wall, and sonar returns have an accuracy within one centimeter.

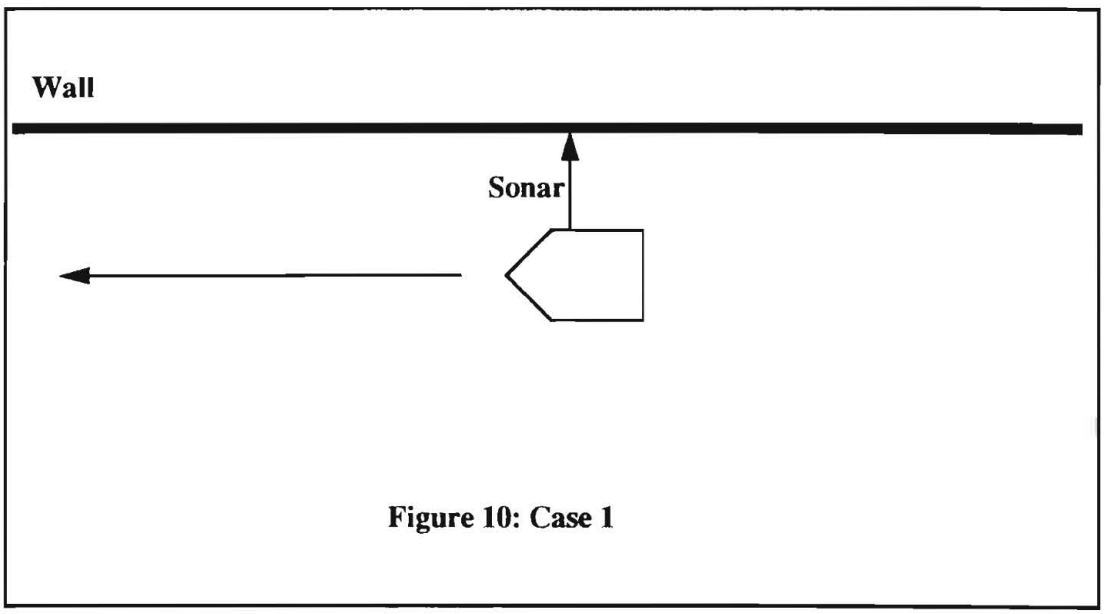

Figure 10: Case 1

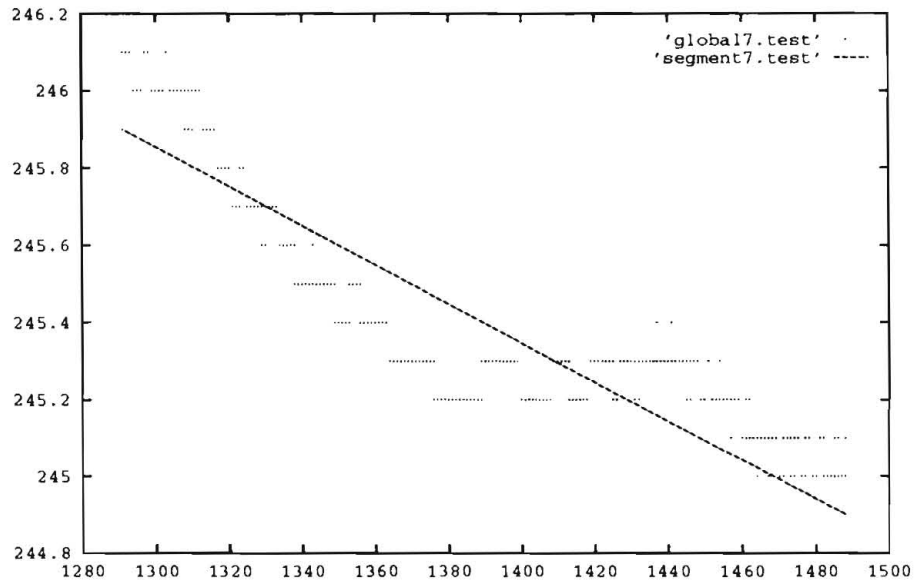

Figure 11: Global and segment sonar data from translational scan

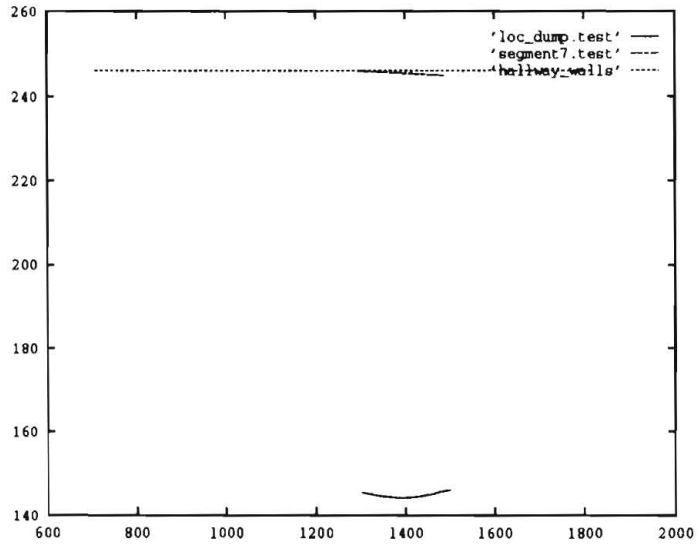

Figure 12: Local trace and segment sonar data from translational scan

B. CASE 2

The robot is moving using its sonar in a translational scan and transfers to a line 90 degrees from its starting line at a corner as in Figure 13.

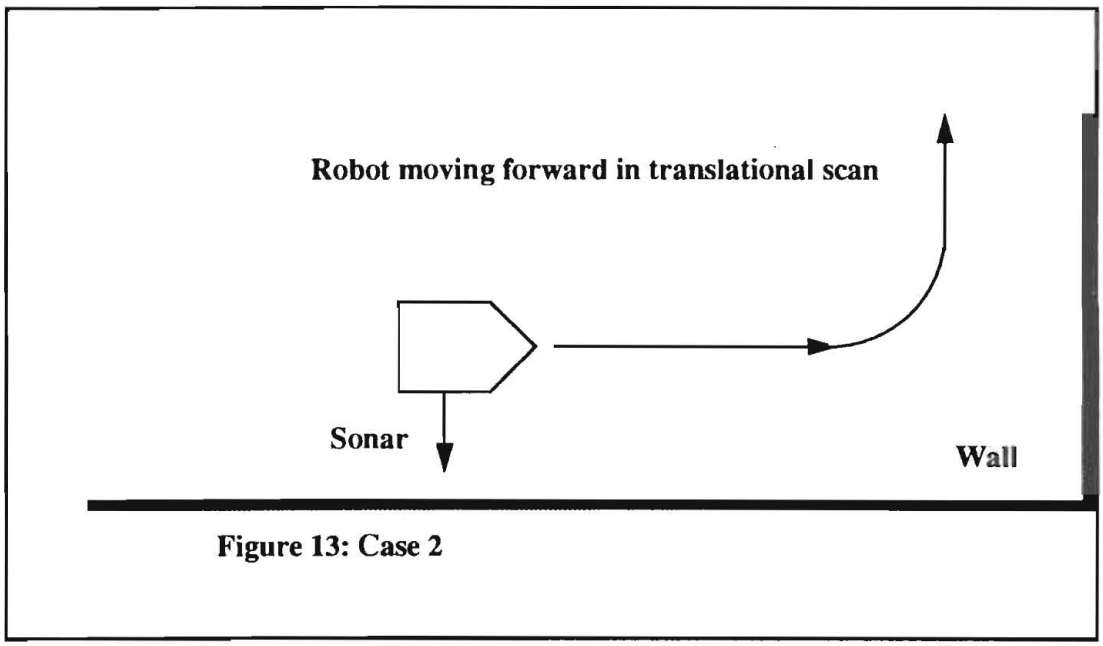

Figure 13: Case 2

The results can be found in Figure 14 and Figure 15. The sonar can accurately detect both walls, with a 45 degree segment produced for the corner.

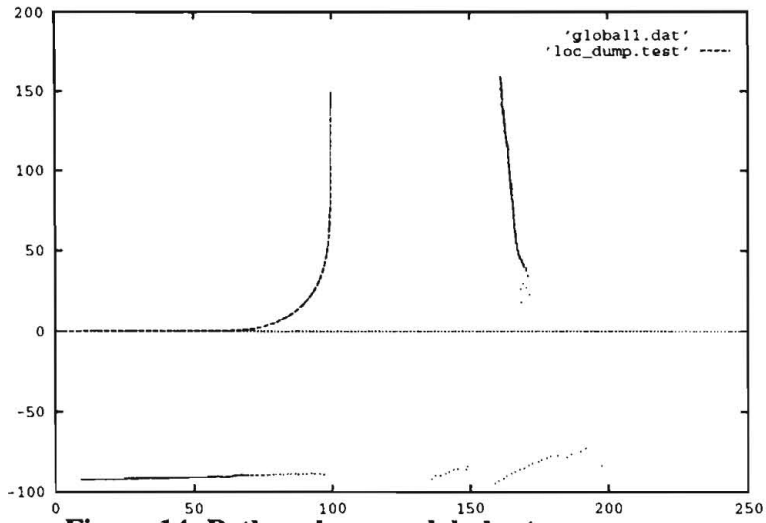

Figure 14: Path and sonar global returns

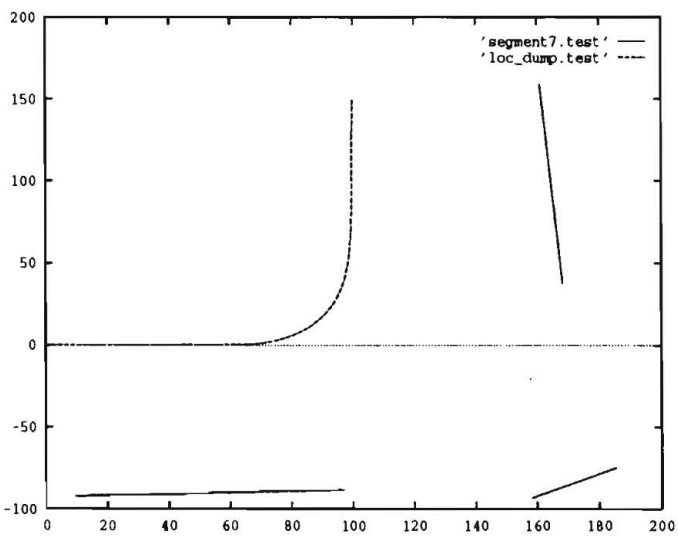

Figure 15: Path and sonar segment data

C. CASE 3

The third case is the robot in a translational scan of a corner and the actual way it perceives it surroundings as in Figure 16. You would expect the results would not accurately reflect the corner due to the amount of reflection, and the poor angle to get returns off the wall. This is the case with the sonar not detecting the walls close to the point where they meet at a 90 degree angle, as shown in Figure 17 and Figure 18.

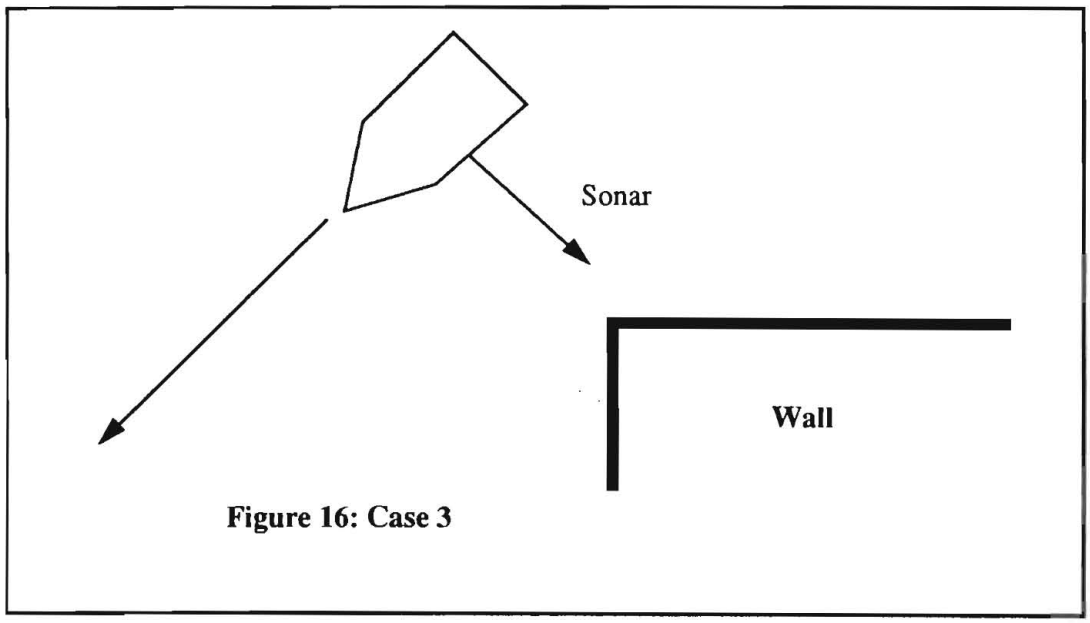

Figure 16: Case 3

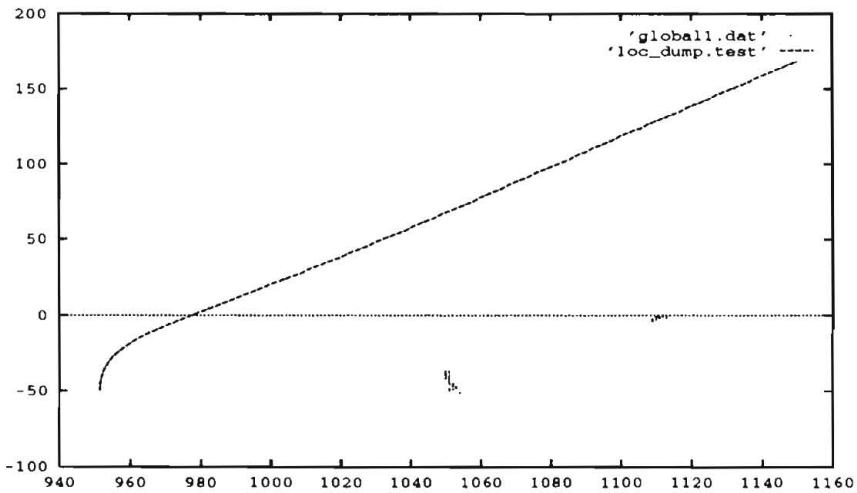

Figure 17: Path and global sonar returns

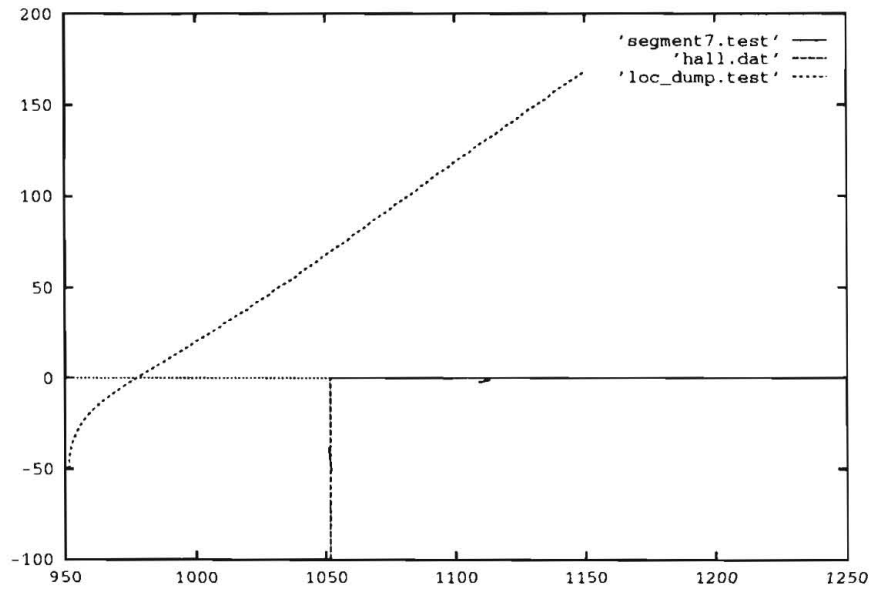

Figure 18: Path, corner, and sonar segment data

D. CASE 4

The fourth case is the robot in a rotational scan as in Figure 19 with a horizontal wall. With this type of obstacle you would expect that the detection rate would be good. Tests results have shown that the robot is able to recognize the wall using the linear square fitting algorithm with line segments, with the robot rotating, as seen in Figure 20 and Figure 21.

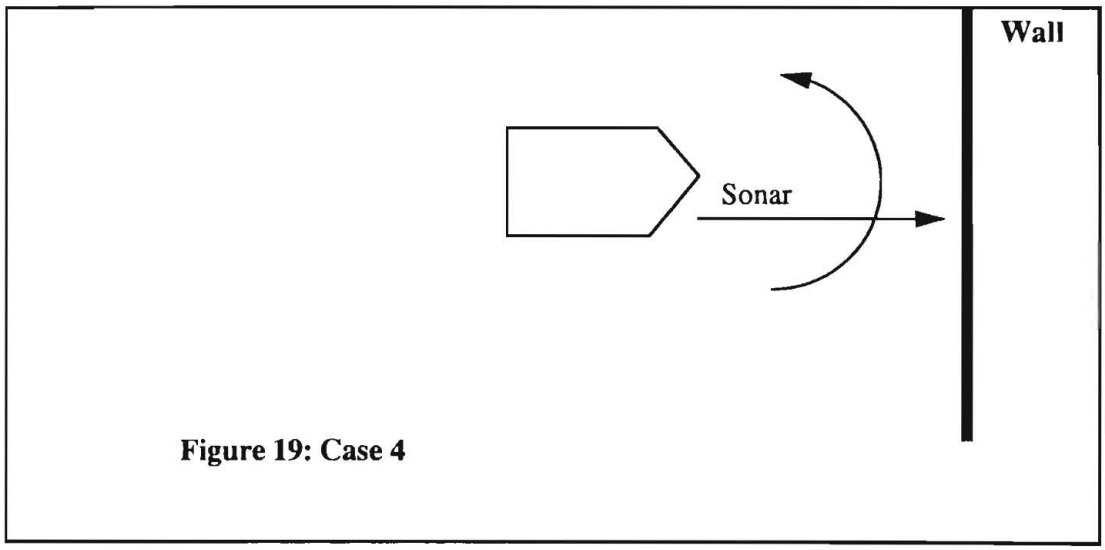

Figure 19: Case 4

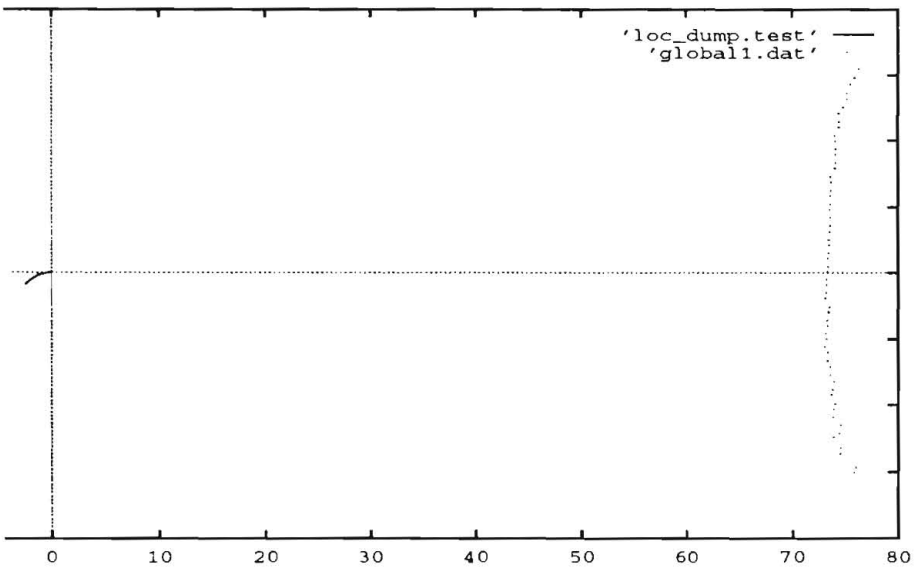

Figure 20: Path and sonar global data

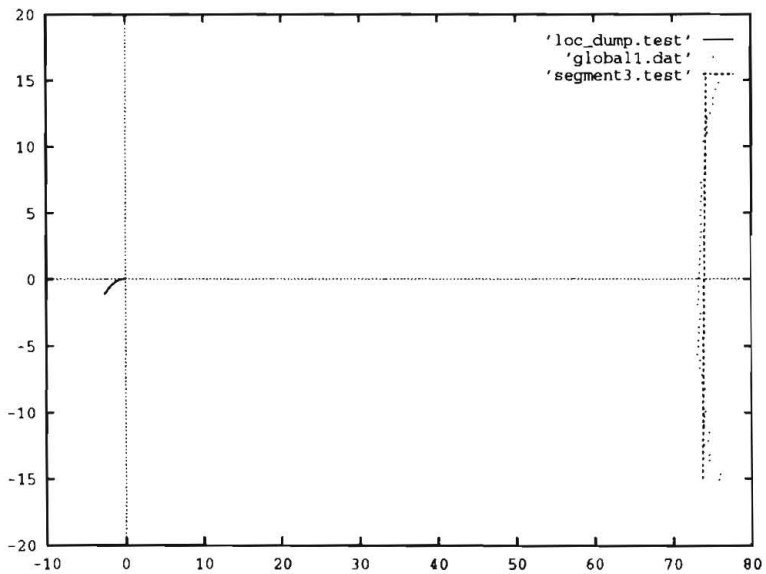

Figure 21: Sonar segment data and hallway

E. CASE 5

The fifth case, seen in Figure 22 is the robot in a rotational scan with two walls forming a 90 degree angle. The results are similar to those in case 2, with a translational scan. The results can be seen in Figure 23 and Figure 24

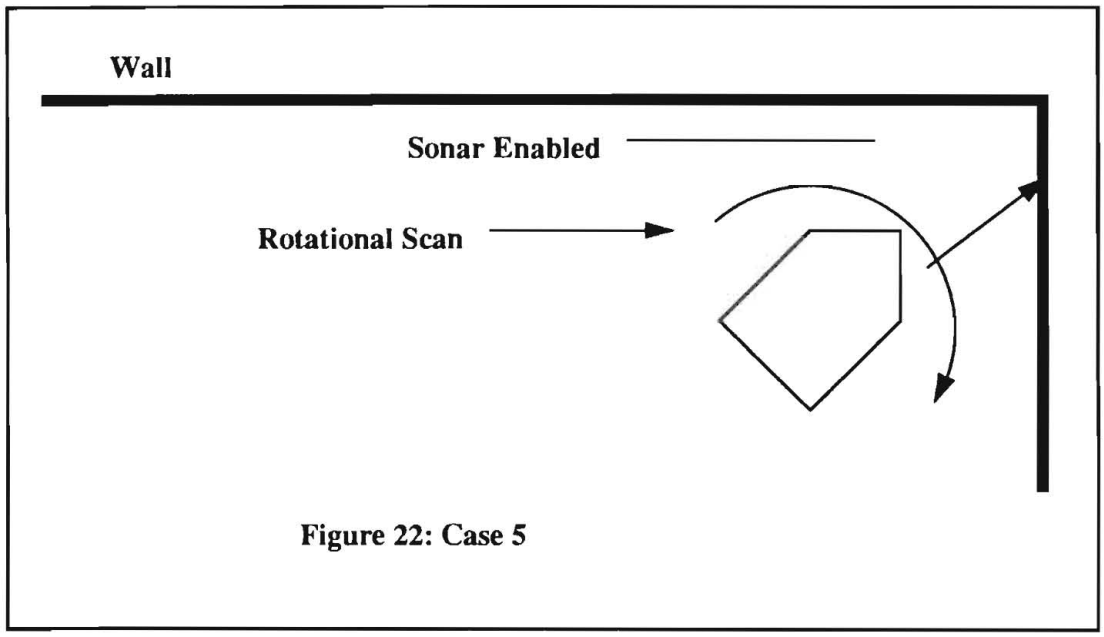

Figure 22: Case 5

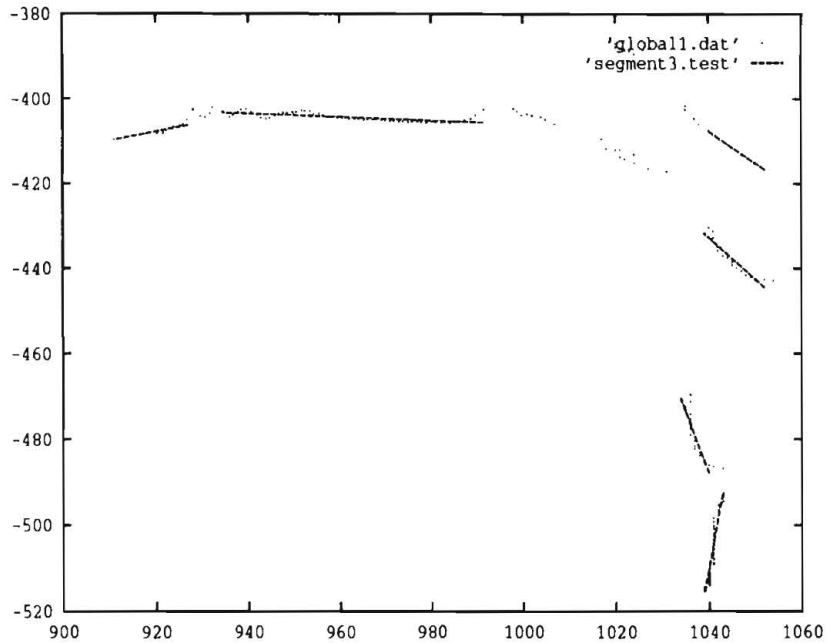

Figure 23: Global and segment sonar data

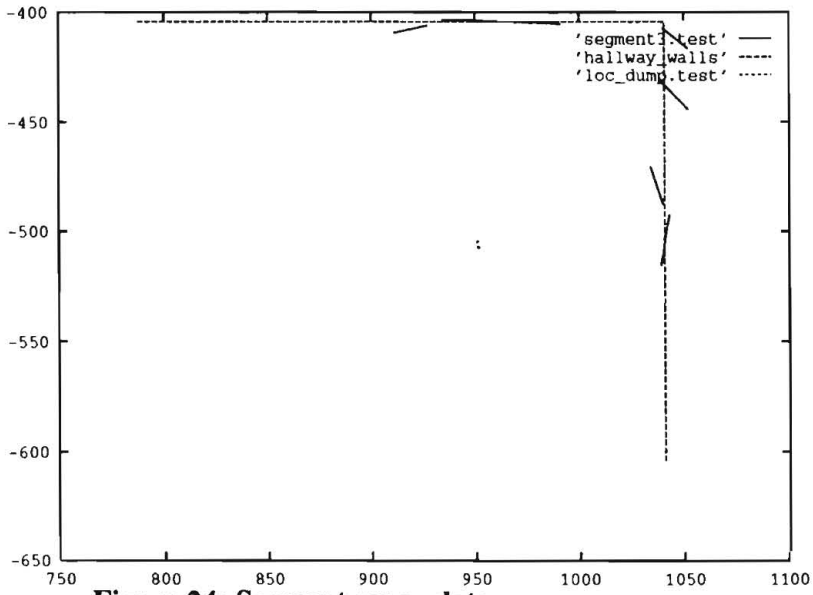

Figure 24: Segment sonar data, hallway, and trace

F. CASE 6

The sixth case, seen in Figure 25 is a rotational scan with 2 walls forming a point. In this case there were no sonar returns in the testing that was done, as would be expected due the poor angle of return from the walls back to the sonar.

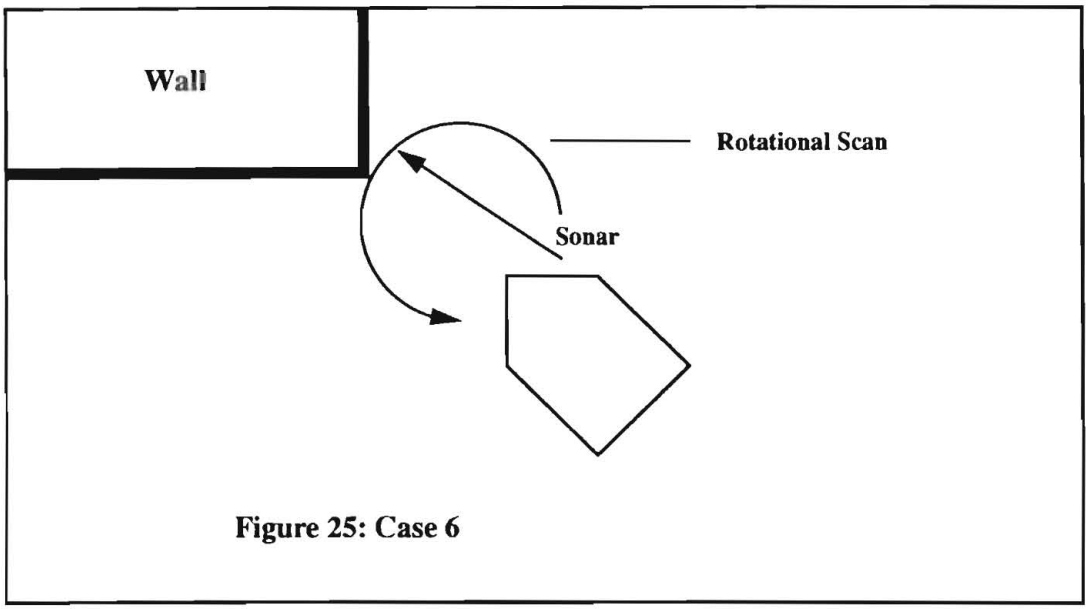

Figure 25: Case 6

G. CASE 7

In this case, seen in Figure 26 we have a circular object, and for testing purposes a plastic can with a 55 centimeter diameter and height of 70 centimeters was used. The purpose was to test how difficult it would be to recognize a circular object. In a translational scan, the circular object was very accurately detected. The global sonar returns in Figure 27 show the curvature of the object. When testing for a segment, we are able to detect a line segment from the obstacle, which will assure the ability to map the obstacle that is there.

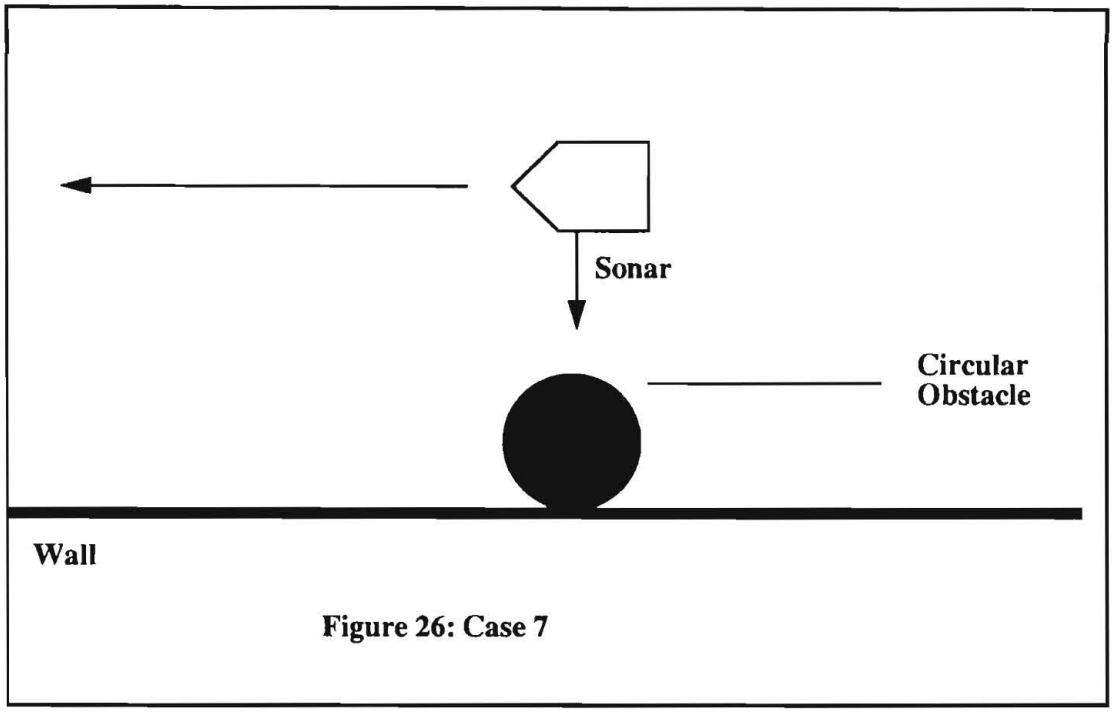

Figure 26: Case 7

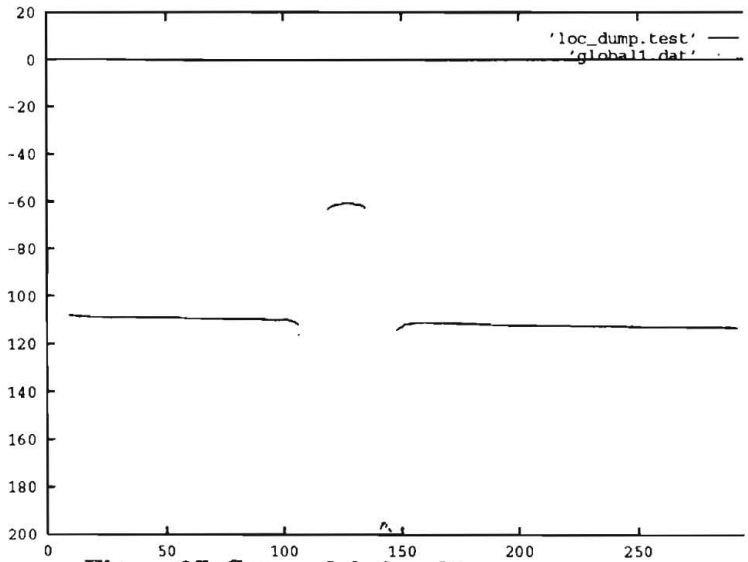

Figure 27: Sonar global and trace

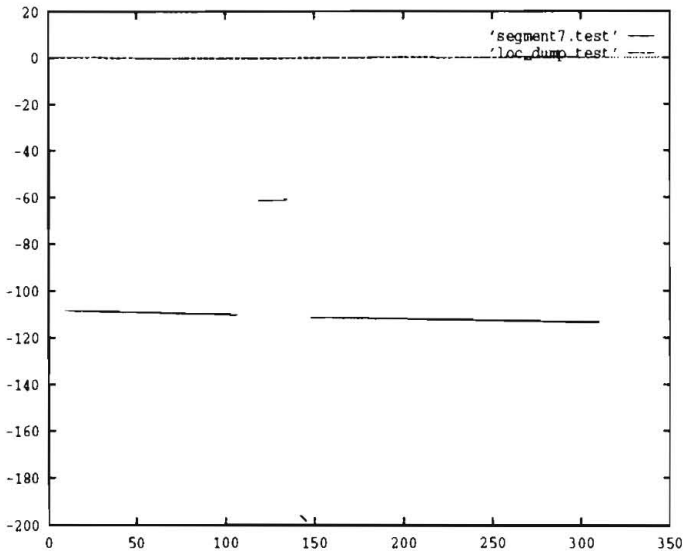

Figure 28: Sonar segment and trace

H. CASE 8

In this case, seen in Figure 29 we have a circular object, and for testing purposes a plastic can with a 55 centimeter diameter and height of 70 centimeters was used. The purpose was to test how difficult it would be to recognize a circular object in a rotational scan. The results are shown in Figure 30 and Figure 31. The obstacle is detected but not as accurately as in the translational scan.

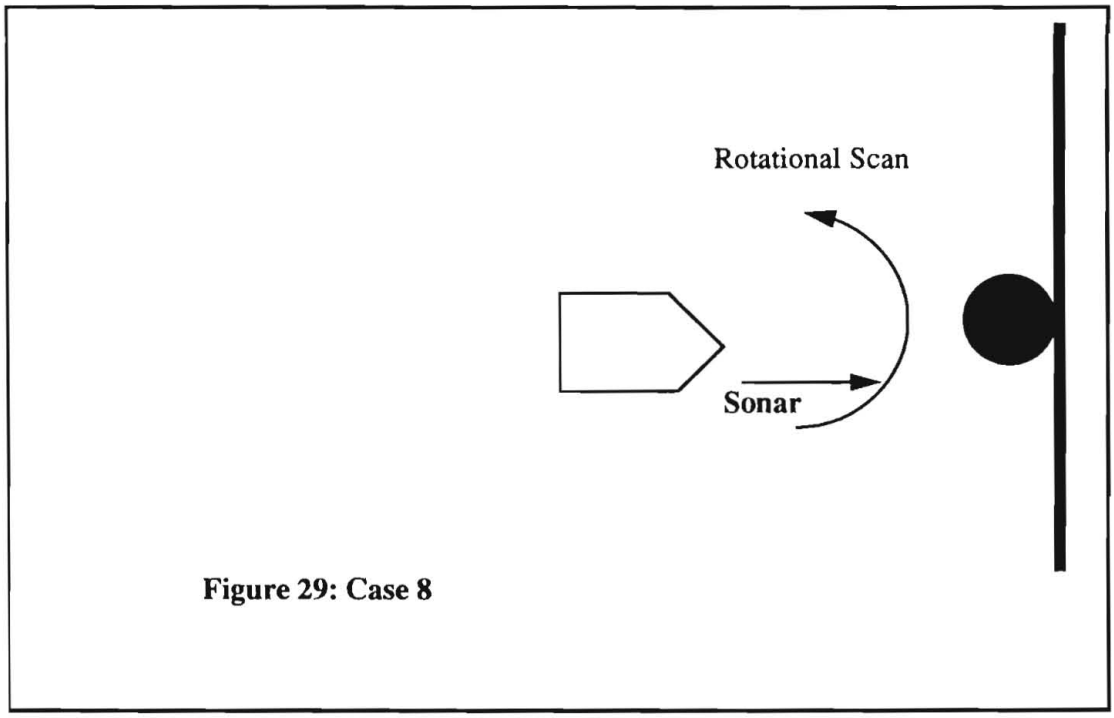

Figure 29: Case 8

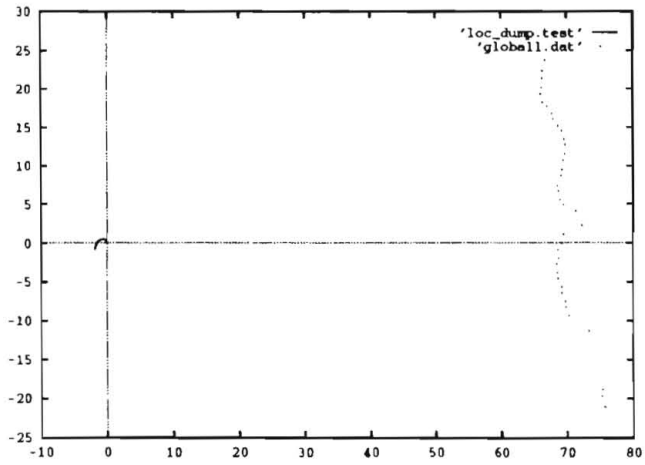

Figure 30: Sonar global and trace

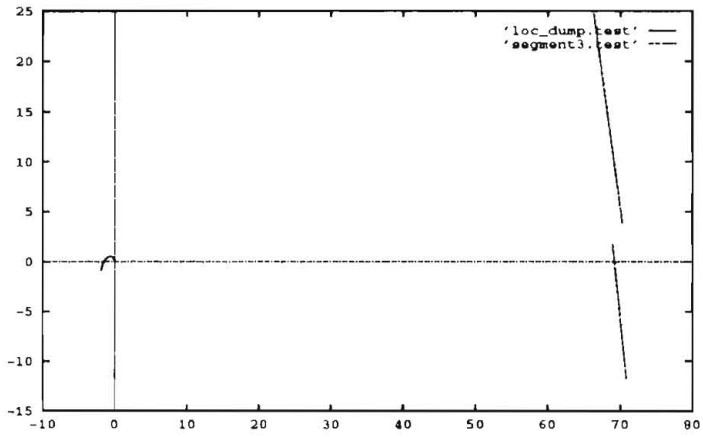

Figure 31: Sonar segment and trace data

VIII. OBSTACLE AVOIDANCE UTILIZING SONARS

A. OBSTACLE AVOIDANCE

Given that you have a starting point, and a finishing point, you should be able to reach your goal even if there is an obstacle in your path. We will start out with simple cases that are easily resolved by the vehicle. The starting point and goal point will be on one line, with the robot transiting to a parallel line when an obstacle is detected. When the compose function previously discussed is used, it is possible to designate a parallel line to the left or the right of *Yamabico*'s current path, with the user either designating the distance between the two paths or using a side sonar to determine the distance of the parallel path. Using the compose function to compute a parallel path allows *Yamabico* to dynamically alter its path.

1. Detecting Obstacle With No Depth

This is the simplest case. There is a starting point and a goal point on a linear line. There is a small obstacle. There is no depth to the obstacle, and once the object is detected, the vehicle will shift to a parallel line to avoid the obstacle, and then shift back to its original path (see figure 32). The general assumptions made in this case are that there is room for the vehicle to maneuver to a parallel line, that there is at most one obstacle and that there is no depth to the obstacle. For this case only one forward looking sonar will be necessary to detect the obstacle.

To detect a small obstacle, the vehicle moves forward with its forward sonar. If an obstacle is detected, the vehicle will maneuver to a parallel line with a specified distance from the original line, left or right of the obstacle. Which way the vehicle turns to avoid the obstacle is left to the user. The user can use side sonars to determine which side will give the robot greater freedom to maneuver. Or for example he can have a heuristic that anytime an obstacle is detected by a forward sonar that he will shift to a parallel line to the right. The distance to shift can be a simple heuristic, for example shift to a parallel line one meter

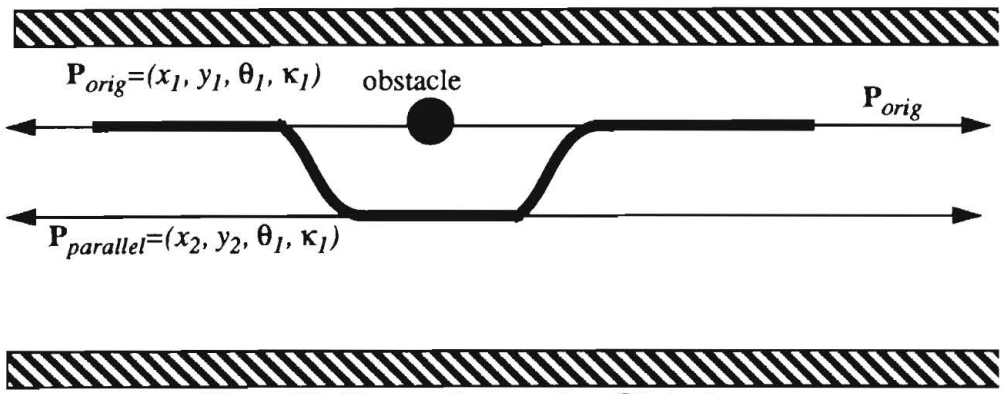

Figure 32: Traversal around an Obstacle

distance to the original path, or it can be determined using side sonars, for example taking a range, subtracting 50 centimeters from it and then shift that distance. A sample pseudo-code program for obstacle avoidance is as follows:

```
define_line(x1, y1, θ1, κ1, &p1)
follow_line(&p1)
if(sonar_detects_forward_obstacle) then
    define_parallel_line(x2, y2, θ1, κ1, &p2)
    shift_to_parallel_line(&p2)
    shift_to_orig_line(x1, y1, θ1, κ1, &p1)
endif
```

Figure 33 shows that the robot has tracked a line Y = 150 with a goal configuration (1650,150) and 0 degree orientation. The robot opens its front sonar while it is tracking on its current path, as soon as the distance from an obstacle is less than 100 centimeters, it transitions to an avoidance path which is line Y = 50. When the robot passes the obstacle, it returns to its original path after traveling past the obstacle for two meters, and stops at its final goal configuration. This can be done with a minimal number of commands. The user.c file used to direct the robot's mission can be found in Appendix B.

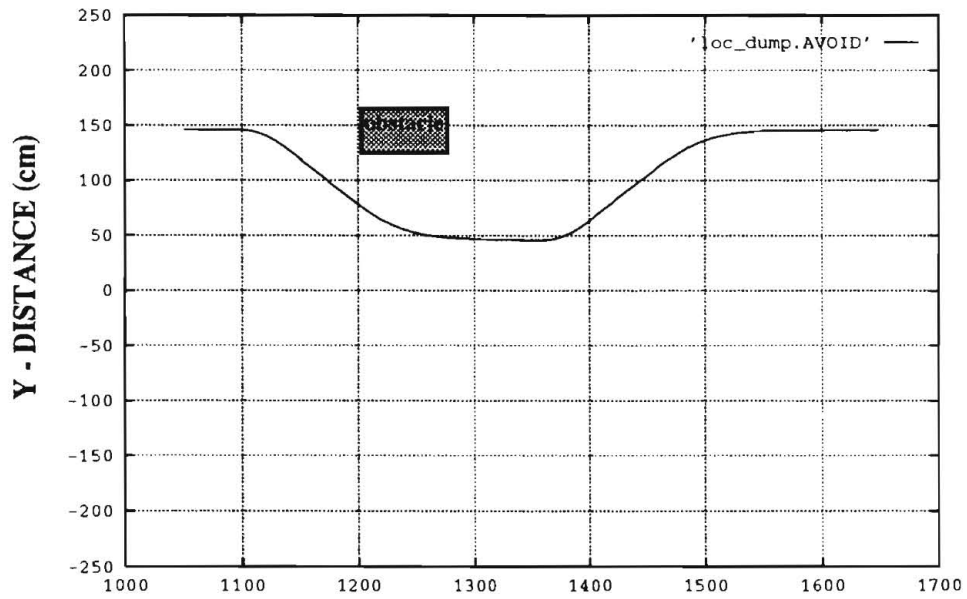

Figure 33: Obstacle avoidance results

2. Detecting Obstacle With Depth

The second case will assume that the obstacle will have depth, and that it is thin enough to maneuver around. There is a start and a finish on a linear line. The depth of the obstacle is unknown, so the vehicle will need to determine that it is safe to resume its original path once the obstacle is clear (see figure 34). The general assumptions made in this case are that there is room for it to maneuver to a parallel line, and there is at most one obstacle. For this case, one forward looking sonar, and side looking sonars will be necessary to detect the obstacle and to detect that it is clear of the obstacle.

To do this, the vehicle is set in motion with a forward sonar and side sonars on. Once an obstacle is detected, the vehicle will shift to a parallel line on its left or right as long as it is safe to do so. If there is an obstacle detected with the side sonars, the vehicle will have to determine whether or not there is enough room to clear the forward obstacle and side obstacle. In our case with *Yamabico*, perhaps we will maneuver to a line one meter

from the obstacle detected. The robot should maneuver to the parallel line with the most room. If there is an obstacle detected within some cut off range on both sides, the vehicle will stop and wait for further instructions.

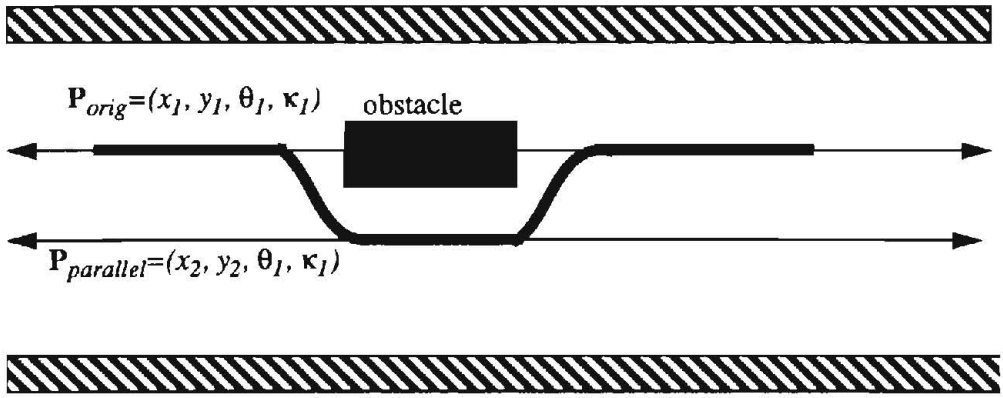

Figure 34: Traversal around an Obstacle with Depth

Once the vehicle transitions to the parallel line, it will detect the obstacle it has shifted lines to avoid. Once the vehicle sonar no longer detects the obstacle, it will shift back to its original line and continue towards its goal point. The pseudo-code program for obstacle avoidance is as follows:

```
define_line(x1, y1, θ1, κ1, &p1)
follow_line(&p1)
if(sonar_detects_forward_obstacle) then
    define_parallel_line(x2, y2, θ1, κ1, &p2)
    shift_to_parallel_line(&p2)
  while(sonar_detects_obstacle_at_side)
    remain on parallel_line
  endwhile
  shift_to_orig_line(x1, y1, θ1, κ1, &p1)
endif
```

Yamabico can use sonar as its environmental sensors to execute this obstacle avoidance missions. Figure 35 shows that the robot has tracked a line Y = 0 with a goal configuration (500,0) and 0 degree orientation. The robot opens its front sonar while it is tracking on its current path, as soon as the distance from an obstacle is less than 100 centimeters, it transitions to an avoidance path which is line Y = -100 and opens the side sonar to detect the obstacle until it passes the obstacle. When the robot passes the obstacle, it returns to its original path and stops at its final goal configuration. This can be done with a minimal number of commands. The user.c file used to direct the robot's mission can be found in Appendix B.

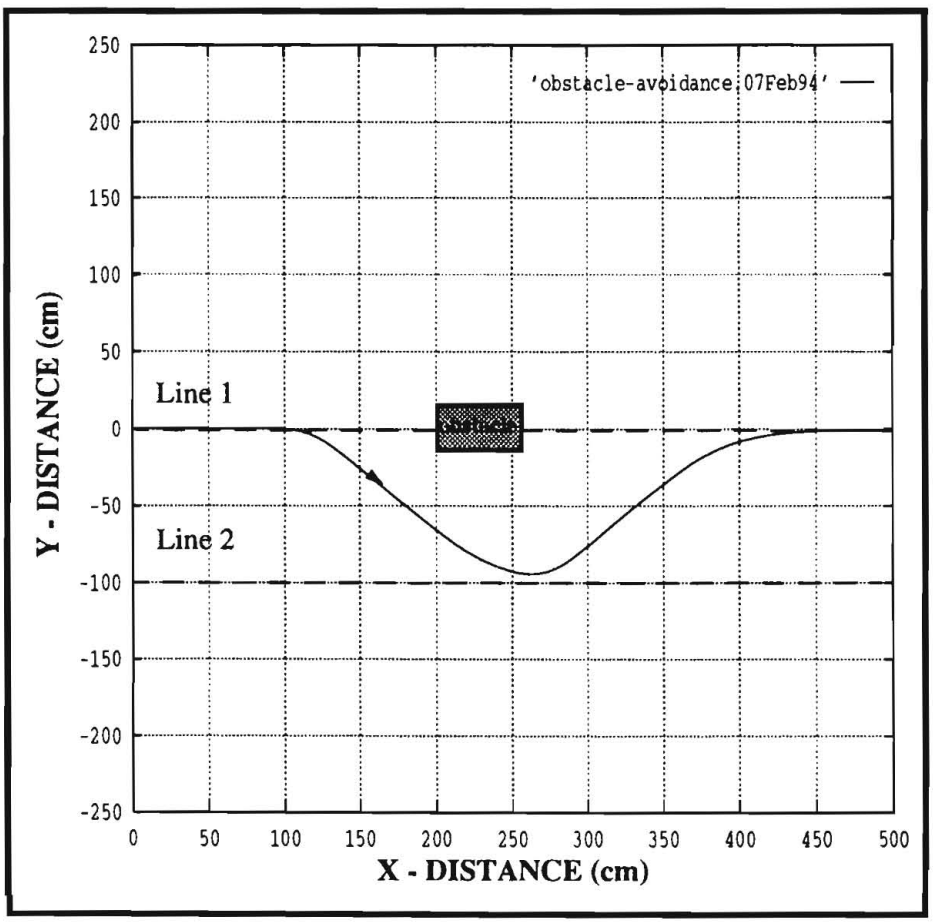

Figure 35: Obstacle avoidance results

IX. CONCLUSION

A. RESULTS

Yamabico's sonar function library for the sonar system is now complete. It accurately applies all algorithms and the results are very accurate. The linear square fitting algorithm is accurately applied to sonar data returned from walls to build line segments that will reflect the wall. The sonars now use the compose function to compute the global position of sonar returns. The user can employ a select number of functions to utilize the sonars in obstacle avoidance.

Basic sonar characteristics taken by translational scanning and rotational scanning showed very reasonable results for the sonars. The experiments taken and results indicate that there is a high degree of accuracy using the sonars while the robot is moving.

Simple obstacle avoidance is a success. The motion system and sonar system coordination is a success. Testing has shown that the motion functions and sonar functions can be jointly used to successfully detect an obstacle, and dynamically alter its path to avoid the obstacle. The coordination is perfect between the sonar and motion systems.

B. RECOMENDATIONS

Some of the 12 sonars have been upgraded and hence are more accurate than others. The front four sonars are the most recently replaced sensors. The remaining sonars need hardware upgrading, and testing should be done to see ensure sonars are working optimally.

For better and more complex avoidance, the use of a parabola is needed in the locomotion functionality. This will improve the transition from one path to another path, and allow more complex motion when avoiding an obstacle. Follow on work needs to ensure that the motion and sonar systems continue to work together.

APPENDIX

This appendix contains the C code for all sonar functions and for the user files that generated the results found in this thesis.

A. SONAR CODE

```
/*
 Author     : Patrick Byrne
 Date       : 22 February 1994
 File       : sonarcard.h
 Description : Provides extern declarations for functions in sonarcard.c
*/
extern void enable_interrupt_operation();
extern void disable_interrupt_operation();
extern void enable_linear_fitting();
extern void disable_linear_fitting();
extern void enable_sonar();
extern void disable_sonar();
extern void serve_sonar();
extern double wait_sonar();
extern void reset_moments();

 * Author   : Patrick Byrne
 * Date     : 22 February 1994
 * File     :
 * sonarcard.c
 * Description : Provides the following functions for the
 * Sonar Interface card in sonarcard.c:
 * void enable_interrupt_operation(); void disable_interrupt_operation(); void
 * enable_linear_fitting(); void disable_linear_fitting(); void
 * enable_sonar(); void disable_sonar(); void serve_sonar(); double
 * wait_sonar(); void reset_moments(); void wait_until();
 *
 */

#include "mml.h"
#ifdef SIM
#include "/n/gemini/work2/yamabico/mml/Sim/spatial.h"
#endif
```

```
/*********************************************************************
 * Procedure:  enable_sonar(n)
 * Description:  enables the sonar group
 * that contains sonar n, which causes all the sonars in that group to
 * echo-range and write data to the data registers on the sonar
 * control board. Marks the n'th  position of the enabled_sonars array
 * to track which sonars are enabled.
*********************************************************************/

void
enable_sonar(n)
   int          n;
{

#ifndef SIM
   int          i;
   i = imaskoff();
#endif
   enabled_sonars[n] = 1;
   switch (n) {
   case 0:
   case 2:
   case 5:
   case 7:
#ifndef SIM
            enabled = enabled | 0x01;
#else
            sonar_group[0] = ON;
#endif
            break;
   case 1:
   case 3:
   case 4:
   case 6:
#ifndef SIM
            enabled = enabled | 0x02;
#else
            sonar_group[1] = ON;
#endif
            break;
   case 8:
   case 9:
   case 10:
```

```
    case 11:
#ifndef SIM
            enabled = enabled | 0x04;
#else
            sonar_group[2] = ON;
#endif
            break;
    case 12:
    case 13:
    case 14:
    case 15:
            enabled = enabled | 0x08;
            break;
    }
#ifndef SIM
    *command_ptr = enabled;
    imaskon(i);
#endif
}

/*********************************************************************
 *
 * Procedure:  disable_sonar(n)
 * Description:  removes the sonar n
 * from the enabled_sonars list. If sonar n is the only enabled sonar
 * from it's group, then the  group is disabled as well and will stop
 * echo ranging. This has benefit of shortening the ping interval for
 * groups that remain enabled.
*********************************************************************/

void
disable_sonar(n)
    int         n;
{
    int         i, c;
#ifndef SIM
    i = imaskoff();
    enabled_sonars[n] = 0;
#endif
    switch (n) {
    case 0:
    case 2:
    case 5:
```

```
    case 7:
            c = enabled_sonars[0] + enabled_sonars[2] +
                    enabled_sonars[5] + enabled_sonars[7];
            if (c == 0)
#ifndef SIM
            enabled = enabled & 0xfe;
#else
            sonar_group[0] = OFF;
#endif
            break;
    case 1:
    case 3:
    case 4:
    case 6:
            c = enabled_sonars[1] + enabled_sonars[3] +
                    enabled_sonars[4] + enabled_sonars[6];
            if (c == 0)
#ifndef SIM
            enabled = enabled & 0xfd;
#else
            sonar_group[1] = OFF;
#endif
            break;
    case 8:
    case 9:
    case 10:
    case 11:
            c = enabled_sonars[8] + enabled_sonars[9] +
                    enabled_sonars[10] + enabled_sonars[11];
            if (c == 0)
#ifndef SIM
            enabled = enabled & 0xfb;
#else
            sonar_group[2] = OFF;
#endif
            break;
    case 12:
    case 13:
    case 14:
    case 15:
            c = enabled_sonars[12] + enabled_sonars[13] +
                    enabled_sonars[14] + enabled_sonars[15];
            if (c == 0)
```

```
#ifndef SIM
            enabled = enabled & 0xf7;
#else
            sonar_group[3] = OFF;
#endif
            break;
    }
#ifndef SIM
    *command_ptr = enabled;
    imaskon(i);
#endif
}

/*****************************************************************
 * Procedure: wait_sonar(n)
 * Description:  waits in a loop until new data is available for
 * sonar n.
 **** ************************************************************/

double
wait_sonar(n)
    int         n;
{
    sonar_table[n].update = 0;
    while (sonar_table[n].update == 0);
    return sonar_table[n].d;
}


/*****************************************************************
 * Procedure: enable_linear_fitting(n)
 * Description:  causes the background system to gather data points
 *     from sonar n and form them into line segments as governed by
 *     the linear fitting algorithm. Increments service_flag.
 ****************************************************************/

void
enable_linear_fitting(n)
    int         n;
{
    sonar_table[n].fitting = 1;
    sonar_table[n].global = 1;
    ++service_flag;
```

```
}

/*********************************************************************/
/* Procedure:  disable_linear_fitting(n)
 * Description:  causes background system to cease forming line
 * segments for sonar n.
 * Decrements the service_flag. Will also disable the calculation of
 * global coordinates for that sonar if data logging of global data is
 * not enabled.
*********************************************************************/

void
disable_linear_fitting(n)
   int          n;
{
   generate_segment(n);
   sonar_table[n].fitting = 0;
   if (sonar_table[n].filetype[1] == 0)
            sonar_table[n].global = 0;
   --service_flag;
}


/*********************************************************************
 * Procedure: enable_interrupt_operation()
 * Description: places sonar
 * control board in interrupt driven mode.
*********************************************************************/

void
enable_interrupt_operation()
{
   *BIM_ptr = *BIM_ptr | 0x10;
}

/*********************************************************************
 * Procedure: disable_interrupt_operation() Description: stops interrupt
 * generation by the sonar control board. A flag is set in the status
 * register when data is ready, and it is the user's responsibility to poll
 * the sonar system for the flag.
*********************************************************************/

void
```

```
disable_interrupt_operation()
{
   *BIM_ptr = *BIM_ptr & 0xef;
}


#ifndef SIM

/*************************************************************************
 * Procedure: serve_sonar(x,y,t,ovfl,data1,data2,data3,data4,group)
 * Description:  this procedure is the "central command" for the
 * control of all sonar related functions. It is linked with the
 * ih_sonar routine and loads sonar data to the sonar_table from
 * there. It then examines the various control flags in the
 * sonar_table to determine which activities the user wishes to take
 * place, and calls the appropriate functions. This procedure is
 * invoked approximately every thirty milliseconds by an interrupt
 * from the sonar control board.
 *************************************************************************/

void
serve_sonar(x, y, t, ovfl, data4, data3, data2, data1, group)
   double     x, y, t;
   int        ovfl, data4, data3, data2, data1, group;
{
    int       n;
   int        i;
   int        data[4];
   int        ovfl_mask = 8;

   data[0] = data1;
   data[1] = data2;
   data[2] = data3;
   data[3] = data4;

   for (i = 0; i < 4; i++, ovfl_mask /= 2) {
          n = group_array[group][i];
          if (ovfl_mask & ovfl){
                 sonar_table[n].d = INFINITY;
          }
          else {
                 sonar_table[n].d = (double) data[i] / 10.0;
          }
```

```
                sonar_table[n].x = x;
                sonar_table[n].y = y;
                sonar_table[n].t = t;
                sonar_table[n].update = 1;
        }

        if (service_flag != 0) {
                for (i = 0; i < 16; i++) {
                        if (sonar_table[i].update == 1) {
                                if (sonar_table[i].global == 1)
                                        calculate_global(i);
                                if (sonar_table[i].fitting == 1)
                                        linear_fitting(i);
                                if (sonar_table[i].filetype[0] == 1)
                                        log_data(i, 1, sonar_table[i].filenumber[0], 0);
                                if (sonar_table[i].filetype[1] == 1)
                                        log_data(i, 2, sonar_table[i].filenumber[1], 0);
                        }
                        sonar_table[i].update = 0;
                }
        }
    }
    #else

    /
**********************************************************************
     Procedure: serve_sonar(w, group)

     Description:  this procedure is the Simulator "central command" for the
        control of all sonar related functions.  It then examines the various
        sonar_table to determine which activities the user wishes to
        control flags in the take place, and calls the appropriate functions.
        This procedure is invoked every third ping.
     **********************************************************************
/
    void
    serve_sonar(w, group)
        World       *w;
        int         group;
    {
    }
    #ifdef jjj
        Line_segment    sonar_line;
```

```
        Line_segment    wall;
        Point           p, q, p1, q1;
        Polygon          *current_poly;
        Vertex           *current_vertex;
        int             i;
        int             j;
        int             k;
        int             n;
        double           sonar_line_theta;
        double           wall_theta;
        double           sonar_theta;

        /* for each sonar in the group being served */
        for (i = 0; i < 4; i++) {
                n = group_array[group][i];

                /* save the robot posture */
                sonar_table[n].x = vehicle.x;
                sonar_table[n].y = vehicle.y;
                sonar_table[n].t = vehicle.t;
                if ((sonar_group[group]) && (enabled_sonars[n])) {
                        /*
                         * printf("%s%d\n","Sonar group firing => ", group);
                         */
                        sonar_table[n].d = INFINITY;

                        /* define the sonar beam */
                        p.x = vehicle.x + (cos(vehicle.t + sonar_table[n].phi) *
sonar_table[n].offset);
                        p.y = vehicle.y + (sin(vehicle.t + sonar_table[n].phi) *
sonar_table[n].offset);
                        q.x = p.x + (cos(vehicle.t + sonar_table[n].axis) * 410.0);
                        q.y = p.y + (sin(vehicle.t + sonar_table[n].axis) * 410.0);
                        sonar_line.p1 = p;
                        sonar_line.p2 = q;
                        sonar_line_theta = orientation(p, q);

                        current_poly = w->poly_list;
                        for (k = 1; k <= w->degree; k++) {
                                current_vertex = current_poly->vertex_list;
                                for (j = 0; j < current_poly->degree; j++) {
                                        p1 = current_vertex->point;
                                        q1 = current_vertex->next->point;
```

```
                                wall.p1 = p1;
                                wall.p2 = q1;
                                wall_theta = orientation(p1, q1);
                                sonar_theta     =     fabs(normalize(wall_theta     -
sonar_line_theta));
                                if ((segment_crossing_test(&wall, &sonar_line) ==
1) &&
                                   (sonar_theta < (HPI + 0.26)) && (sonar_theta >
(HPI - 0.26)))
                                /*
                                 * wall and sonar beam must
                                 * intersect at 90 +- 15
                                 * degrees
                                 */
                                {
                                if (wall_theta == 0.0 || wall_theta == PI) {
                                if   (sonar_table[n].d   >   fabs((p.y   -   p1.y)   /
sin(sonar_theta))) {
                                sonar_table[n].d    =    fabs((p.y    -    p1.y)    /
sin(sonar_theta));
                                sonar_table[n].update = 1;
                                }
                                };
                                if (wall_theta == -HPI || wall_theta == HPI) {
                                if   (sonar_table[n].d   >   fabs((p.x   -   p1.x)   /
sin(sonar_theta))) {
                                sonar_table[n].d    =    fabs((p.x    -    p1.x)    /
sin(sonar_theta));
                                sonar_table[n].update = 1;}
                                };
                                }
                                current_vertex = current_vertex->next;
                        }       /* end for each vertex loop */

                }       /* end for each polygon loop */
        }               /* end if sonar is enabled test */
}                       /* end outer for each sonar in group loop */
/*
 * printf("%s%d%s%2.2lf\n","Sonar ", n," Range is  => ",
 * sonar_table[n].d);
 */
if (service_flag != 0)
        for (i = 0; i < 16; i++) {
```

```
                if (sonar_table[i].update == 1) {
                        if (sonar_table[i].global == 1)
                                calculate_global(i);
                        if (sonar_table[i].fitting == 1)
                                linear_fitting(i);
                        if (sonar_table[i].filetype[0] == 1)
                                log_data(i, 1, sonar_table[i].filenumber[0], 0);
                        if (sonar_table[i].filetype[1] == 1)
                                log_data(i, 2, sonar_table[i].filenumber[1], 0);
                }       /* end if */
                sonar_table[i].update = 0;
        }               /* end for each sonar updated */

}                       /* end serve_sonar */
#endif
#endif
```

```
/*declaration of functions and return values*/
/*sonarmath.h                                   */
extern double sonar();
extern void linear_fitting();
extern posit global();
extern LINE_SEG *get_segment();
extern void calculate_global();
extern void add_to_segment();
extern void generate_segment();
extern LINE_SEG *get_current_segment();
extern LINE_SEG *end_segment();
extern void initialize_sonar();


/*sonarmath.c       */
#include "mml.h"

void
initialize_sonar()
{
   int          i, k;

   /* initialize sonar_table and segment_data */


   for (i = 0; i < 16; i++) {
            sonar_table[i].global = 0;
            sonar_table[i].fitting = 0;
            sonar_table[i].filetype[0] = 0;
            sonar_table[i].filetype[1] = 0;
            sonar_table[i].filetype[2] = 0;
            sonar_table[i].update = 0;
            sonar_table[i].d = 0.0;
            sonar_table[i].x = 0.0;
            sonar_table[i].y = 0.0;
            sonar_table[i].t = 0.0;

            segment_data[i].alpha = 0.0;
            segment_data[i].r = 0.0;
            segment_data[i].startx = 0.0;
            segment_data[i].starty = 0.0;
            segment_data[i].endx = 0.0;
            segment_data[i].endy = 0.0;
```

```
        enabled_sonars[i] = 0;
        segment_data[i].n = 0;

}

/* set up compensation for sonar position */

sonar_table[0].rob_t = 0.0;
sonar_table[1].rob_t = 3.142;
sonar_table[2].rob_t = 3.142;
sonar_table[3].rob_t = 0.0;
sonar_table[4].rob_t = 1.571;
sonar_table[5].rob_t = 1.571;
sonar_table[6].rob_t = -1.571;
sonar_table[7].rob_t = -1.571;
sonar_table[8].rob_t = 2.356;
sonar_table[9].rob_t = -2.356;
sonar_table[10].rob_t = -0.785;
sonar_table[11].rob_t = 0.785;
sonar_table[12].rob_t = 0.0;
sonar_table[13].rob_t = 1.5708;
sonar_table[14].rob_t = 4.7124;
sonar_table[15].rob_t = 0.0;

sonar_table[0].rob_x = 18.0;
sonar_table[1].rob_x = -18.0;
sonar_table[2].rob_x = -18.0;
sonar_table[3].rob_x = 18.0;
sonar_table[4].rob_x = 9.5;
sonar_table[5].rob_x = -9.5;
sonar_table[6].rob_x = -9.5;
sonar_table[7].rob_x = 9.5;
sonar_table[8].rob_x = -15.0;
sonar_table[9].rob_x = -14.5;
sonar_table[10].rob_x = 15.5;
sonar_table[11].rob_x = 16.0;
sonar_table[12].rob_x = 0.0;
sonar_table[13].rob_x = 1.5708;
sonar_table[14].rob_x = 4.7124;
sonar_table[15].rob_x = 0.0;

sonar_table[0].rob_y = 9.5;
sonar_table[1].rob_y = 9.5;
```

```
    sonar_table[2].rob_y = -9.5;
    sonar_table[3].rob_y = -9.5;
    sonar_table[4].rob_y = 19.75;
    sonar_table[5].rob_y = 19.75;
    sonar_table[6].rob_y = -19.75;
    sonar_table[7].rob_y = -19.75;
    sonar_table[8].rob_y = 19.75;
    sonar_table[9].rob_y = -19.75;
    sonar_table[10].rob_y = -19.75;
    sonar_table[11].rob_y = 19.75;
    sonar_table[12].rob_y = 0.0;
    sonar_table[13].rob_y = 21.5;
    sonar_table[14].rob_y = 21.5;
    sonar_table[15].rob_y = 0.0;


    group_array[0][0] = 0;
    group_array[0][1] = 5;
    group_array[0][2] = 2;
    group_array[0][3] = 7;
    group_array[1][0] = 3;
    group_array[1][1] = 4;
    group_array[1][2] = 1;
    group_array[1][3] = 6;
    group_array[2][0] = 10;
    group_array[2][1] = 11;
    group_array[2][2] = 8;
    group_array[2][3] = 9;
    group_array[3][0] = 12;
    group_array[3][1] = 13;
    group_array[3][2] = 14;
    group_array[3][3] = 15;


    service_flag = 0;
    C1 = 0.02;
    C2 = 5.0;


}

/*****************************************************************
 * Procedure:  sonar(n)
```

```
 * Description:  returns the distance (in
 * centimeters) sensed by the  n_th ultrasonic sensor.  If no echo is
 * received, then INFINITY(1.0e6) is  returned.  If the distance is less than 10
 * cm, then a 0 is returned.
**********************************************************************/

double
sonar(n)
   int          n;
{
   return sonar_table[n].d;
}

/**********************************************************************
 * Procedure: global(n)
 * Description: returns a structure of type
 * posit containing the global x and y coordinates of the position of
 * the last sonar return.
**********************************************************************/

posit global   (n)
   int          n;
{
   posit         answer;

   if (sonar_table[n].global == 0)
            calculate_global(n);
   answer.gx = sonar_table[n].gx;
   answer.gy = sonar_table[n].gy;
   return answer;
}

/**********************************************************************
 * Procedure: get_segment(n)
 * Description:  returns a pointer to the
 * oldest segment on the linked   list of segments for sonar n; i.e. the
 * record at the head  of the linked list. It is destructive, thus
 * subsequent calls will return subsequent segments until the list is
 * empty.  This is accomplished by first copying the contents of the
 * head record into a temporary record called segstruct and then
 * freeing the allocated memory for the head record. The pointer
 * returned is actually a pointer to this temporary storage. If
 * get_segment is called on an empty list a null pointer is returned.
```

```
**********************************************************************/

LINE_SEG    *
get_segment(n)
   int          n;
{
   LINE_SEG     *ptr;
   int          index;

   index = seg_list_head[n];
   if (index == -1)
          ptr = NULL;
   else {
          ptr = &seg_list[n][index];
          seg_list_head[n] = (index < 4) ? (index + 1) : 0;
   }
   return ptr;
}

/**********************************************************************
 * Procedure: end_segment(n)
 * Description: this procedure allocates
 * memory for the segment data structure, loads the correct values
 * into it and returns a pointer to the structure.
**********************************************************************/

LINE_SEG    *
end_segment(n)
   int          n;
{
   LINE_SEG     *seg_ptr;
   double       startx, starty, endx, endy, delta, alpha, r, length;

   seg_ptr = &segstruct;

   startx = segment_data[n].startx;
   starty = segment_data[n].starty;
   endx = segment_data[n].endx;
   endy = segment_data[n].endy;
   alpha = segment_data[n].alpha;
   r = segment_data[n].r;
   delta = startx * cos(alpha) + starty * sin(alpha) - r;
   startx = startx - (delta * cos(alpha));
```

```
    starty = starty - (delta * sin(alpha));
    delta = endx * cos(alpha) + endy * sin(alpha) - r;
    endx = endx - (delta * cos(alpha));
    endy = endy - (delta * sin(alpha));
    length = sqrt(SQR(startx - endx) + SQR(starty - endy));

    seg_ptr->headx = startx;
    seg_ptr->heady = starty;
    seg_ptr->tailx = endx;
    seg_ptr->taily = endy;
    seg_ptr->alpha = alpha;
    seg_ptr->r = r;
    seg_ptr->length = length;
    seg_ptr->sonar = n;

    return seg_ptr;
}


/*********************************************************************
 * Procedure: get_current_segment(n)
 * Description:  returns a pointer
 * to the segment currently under construction if there is one,
 * otherwise returns null pointer. This is accomplished by calling
 * end_segment, copying the data into segstruct and then returning a
 * pointer to segstruct. The memory allocated by end_segment is then
 * freed.
*********************************************************************/

LINE_SEG      *
get_current_segment(n)
    int           n;
{
    LINE_SEG      *ptr;


    ptr = end_segment(n);
    return ptr;
}

/*********************************************************************
 * Procedure: calculate_global(n)
 * Description: this procedure
```

```
 * calculates the global x and y coordinates for the range value and
 * robot configuration in the sonar table. The results are stored in
 * the sonar table.
*****************************************************************/

void
calculate_global(n)
   int          n;
{
   double       lx, ly, lt, range, rob_t, rob_x, rob_y;
   CONFIGURATION global;
   range = sonar_table[n].d;
   if (range >= INFINITY0) {
            sonar_table[n].gx = INFINITY;
            sonar_table[n].gy = INFINITY;
   } else {

            rob_x = sonar_table[n].rob_x;
            rob_y = sonar_table[n].rob_y;
            rob_t = sonar_table[n].rob_t;

            get_rob0(&global);


            /* vehicle compose sonar */
            lx = global.   x + (cos(global.t) *rob_x) - (rob_y * sin(global.t));
            ly = global.   y + (sin(global.t) *rob_x) + (rob_y * cos(global.t));
            lt = rob_t + global.t;

            /* vehicle compose sonar range */
            sonar_table[n].gx = lx + (cos(lt) * range);
            sonar_table[n].gy = ly + (sin(lt) * range);
   }
}

/*****************************************************************
 * Procedure: add_to_segment(n, x, y)  * Description: this procedure
 * calculates new interim data for the line segment and stores it in
 * segment_data[n]. It also changes the end point values to the point
 * being added.
*****************************************************************/

void
```

```
add_to_segment(n, x, y)
   int         n;
   double      x, y;
{
   double      m00, m10, m01, m20, m11, m02;
   double      alpha, r;
   double      mux, muy, mm20, mm11, mm02;

   m00 = segment_data[n].m00 += 1.0;
   m10 = segment_data[n].m10 += x;
   m01 = segment_data[n].m01 += y;
   m20 = segment_data[n].m20 += sqr(x);
   m11 = segment_data[n].m11 += x * y;
   m02 = segment_data[n].m02 += sqr(y);

   if (m00 < 1.5) {
         segment_data[n].startx = x;
         segment_data[n].starty = y;
   }
   mux = m10 / m00;
   muy = m01 / m00;
   mm20 = m20 - sqr(m10) / m00;
   mm11 = m11 - m10 * m01 / m00;
   mm02 = m02 - sqr(m01) / m00;

   if (m00 > 1.5) {
         alpha = atan2(-2.0 * mm11, (mm02 - mm20)) / 2.0;
         r = mux * cos(alpha) + muy * sin(alpha);

         segment_data[n].alpha = alpha;
         segment_data[n].r = r;
         segment_data[n].endx = x;
         segment_data[n].endy = y;
   }
}


/*****************************************************************
 * Procedure: reset_moments(n);
 * Description: resets the accumulative
 * values in segment_data[n] /* (m00,m10,m01,m20,m11,m02) to zero.

*****************************************************************/
```

```
void
reset_moments(n)
   int          n;
{
   segment_data[n].m00 = 0.0;
   segment_data[n].m10 = 0.0;
   segment_data[n].m01 = 0.0;
   segment_data[n].m20 = 0.0;
   segment_data[n].m11 = 0.0;
   segment_data[n].m02 = 0.0;
}


/*********************************************************************
 * Procedure: generate_segment(n)
 * Description: this function
 * completes segments at the end of a data run. Necessary because the
 * linear fitting function only terminates a segment based on the data
 * - it has no way of knowing that the user has stopped collecting data.
 *********************************************************************/

void
generate_segment(n)
   int          n;
{
   LINE_SEG     *seg_ptr;
   if (segment_data[n].m00 > 10.0) {
            seg_ptr = end_segment(n);
            build_list(seg_ptr, n);
   }
   reset_moments(n);
}

/*********************************************************************
 * Procedure: linear_fitting(n)
 *       Revised by Y. Kanayama,07-07-93
 * Description: this procedure controls the fitting of point
 * data to straight line segments. First it tests if the new coming
 * point is not far from the fitted line.  If the test is passed, the
 * point is added to test if the thinnes test is passed.  If it is
 * passed, the addition is finalized. If any of the tests fail,
 * the line segment is ended and a new one started. The completed line
```

```
 * segment is stored in a data structure called segment, and segments
 * are linked together in a linked list.
*****************************************************************/

void
linear_fitting(n)
   int          n;
{

   double       x, y, m00;
   double       alpha, r, delta;
   double       sonar_range;
   LINE_SEG     *finished_segment;
   sonar_range = sonar_table[n].d;

   if (sonar_range < 9.3 || sonar_range > 409.0) {
            generate_segment(n);
            return;
   }
   x = sonar_table[n].gx;/* temporary moments     */
   y = sonar_table[n].gy;
   m00 = segment_data[n].m00;

   if (m00 < 1.5) {
            add_to_segment(n, x, y);
            return;
   }
   r = segment_data[n].r;
   alpha = segment_data[n].alpha;
   delta = fabs(r - x * cos(alpha) - y * sin(alpha));

   if (delta > max2(C2, C1 * sonar_range)) {
            generate_segment(n);
            add_to_segment(n, x, y);
            return;
   } else {
            add_to_segment(n, x, y);
            return;
   }

}                           /* end linear_fitting */

/*****************************************************************
```

```
 * Procedure: build_list(ptr, n);
 * Description: this function accepts
 * a pointer to a segment data structure and a sonar number, and
 * appends the segment structure to the tail of a linked list of
 * structures for that sonar.
**********************************************************************/

void
build_list(ptr, n)
   int          n;
   LINE_SEG     *ptr;
{
   int          next;

   if (seg_list_tail[n] == -1)
            seg_list_head[n] = 0;
   next = (seg_list_tail[n] < 4) ? ++seg_list_tail[n] : 0;
   if (next == seg_list_head[n])
            seg_list_head[n] = (seg_list_head[n] < 4) ? ++seg_list_head[n] : 0;
   seg_list[n][next] = *ptr;
   if (sonar_table[n].filetype[2] == 1)
            log_data(n, 3, sonar_table[n].filenumber[2], next);
}
```

```
/*
 Author     : Patrick Byrne
 Date       : February 22, 1994
 File       : sonario.h
 Description : Provides extern declarations for functions in sonario.c
*/

extern void xfer_raw_to_host();
extern void xfer_global_to_host();
extern void xfer_segment_to_host();
extern void host_xfer();

/*
 * Author     : Patrick Byrne
 * Date       : 22 February 1994
 * File       :
 * sonario.c
 * Description : Provides the following functions for sonar io:
 * host_xfer() xfer_segment_to_host() xfer_raw_to_host() xfer_global_to_host()
 */


#include "mml.h"
#ifndef SIM


/*********************************************************************
 *  Procedure: xfer_raw_to_host(filenumber,filename)
 * Description:
 * this function allocates memory for a buffer and then converts a raw data
 * log file to a string format stored in the buffer. It then calls host_xfer
 * to send the string to the host. When that transfer is complete, it
 * frees the memory it allocated for the buffer. Filename must be
 * entered in double quotes ( "dumpraw" for example).
*********************************************************************/

void
xfer_raw_to_host(filenumber, filename)
    int         filenumber;
    char        *filename;
{
    char        *rbuffer;
    char        *start;
```

```
    int         i, c, j;

    i = raw_data_log[filenumber].next;
    c = 20 + (i * 33);
    rbuffer = malloc(c);
    start = rbuffer;
    for (j = 0; j < i; j++) {
            print_flex(raw_data_log[filenumber].darray[j], rbuffer);
            print_flex(raw_data_log[filenumber].xarray[j], rbuffer);
            print_flex(raw_data_log[filenumber].yarray[j], rbuffer);
            print_flex(raw_data_log[filenumber].tarray[j], rbuffer);
            nl_flex(rbuffer);
    }
    putb('\0', rbuffer);
    rbuffer = start;
    host_xfer(rbuffer, filename);
    free(rbuffer);
}


/*********************************************************************
 * Procedure: xfer_global_to_host(filenumber,filename) Description:
 * this function performs the same function as xfer_raw_to_host, but for
 * global data vice raw data.

*********************************************************************/

void
xfer_global_to_host(filenumber, filename)
    int         filenumber;
    char        *filename;
{
    char        *gbuffer;
    char        *start;
    int         i, c, j;


    i = global_data_log[filenumber].next;
    c = 20 + (i * 17);

    /*c = 20 + (i * 22);*/
```

```
    gbuffer = malloc(c);

    start = gbuffer;
    for (j = 0; j < i; j++) {
            print_flex(global_data_log[filenumber].xarray[j], gbuffer);
            print_flex(global_data_log[filenumber].yarray[j], gbuffer);
            nl_flex(gbuffer);
    }
    putb('\0', gbuffer);
    gbuffer = start;
    host_xfer(gbuffer, filename);
    free(gbuffer);
}
/*************************************************************************
 * Procedure: xfer_segment_to_host(filenumber,filename)
 * Description:
 * this function performs the same function as xfer_raw_to_host, but for
 * segment data vice raw data.
 *************************************************************************/

void
xfer_segment_to_host(filenumber, filename)
    int         filenumber;
    char        *filename;
{
    char        *segbuffer;
    char        *start;
    int         i, c, j;

    i = segment_data_log[filenumber].count;
    /* c = 20 + (i * 77); */
    c = 20 + (i * 85);
    segbuffer = malloc(c);
    start = segbuffer;
    for (j = 0; j < i; j++) {
            print_flex(segment_data_log[filenumber].array[j].headx, segbuffer);
            print_flex(segment_data_log[filenumber].array[j].heady, segbuffer);
            nl_flex(segbuffer);
            print_flex(segment_data_log[filenumber].array[j].tailx, segbuffer);
            print_flex(segment_data_log[filenumber].array[j].taily, segbuffer);
            nl_flex(segbuffer);
            nl_flex(segbuffer);
    }
```

```
    putb('\0', segbuffer);
    segbuffer = start;
    host_xfer(segbuffer, filename);
    free(segbuffer);
}


/*************************************************************************
 * Procedure: host_xfer(buffer,filename)
 * Description: this function
 * transfers a data string from the buffer to the host. Not a user
 * function; is called by data conversion functions such as xfer_raw_to_host.
 * User would call the xfer_raw_to_host (or equivalent for global or
 * segment data)  to download data from the robot.

*************************************************************************/

void
host_xfer(buffer, filename)
    char        *buffer;
    char        *filename;
{
    i_port(HOST, PORT_SPEED, 0, 0, 0);
    r_printf("\12\15 connect cable and keyin\" \"");
    while (r_getchar() != ' ');
    putstr("\n", HOST);
    i_port(HOST, PORT_SPEED, 0, 0, 1);
    r_printf("\12\15 ready for dump ");
    while (r_getchar() != 'g');
    putstr("ytof ", HOST);
    putstr(filename, HOST);
    putstr(" w \n", HOST);
    while (r_getchar() != ' ');
    r_printf("dumping ");
    putstr(buffer, HOST);
    putb('\4', HOST);
    putb('\4', HOST);
    r_printf("\7\7\7");
    return;
}

#endif
```

```
/* Includes from sonar sub-modules */
#include "sonarmath.h"
#include "sonarcard.h"
#include "sonario.h"
#include "sonarlog.h"


/*declaration of functions and return values for sonarsys.c*/

extern void set_sonar_parameters();
extern void build_list();
extern CONFIGURATION get_sonar_config();

#include "mml.h"
#ifdef SIM
#include "/n/gemini/work2/yamabico/mml/Sim/spatial.h"
#endif

/*********************************************************************
 * Procedure: set_sonar_parameters(c1,c2)
 * Description:  allows the user to
 * adjust constants which control the  linear fitting algorithm. C1 is
 * a multiplier to allow more lenancy for greater sonar ranges.
 * C2 is an absolute value; both are used to determine if an
 * individual data point is usable for the algorithm.  Default values
 * are set in main.c to .02, 5.0 respectively.
*********************************************************************/

void
set_sonar_parameters(c1, c2)
   double       c1;
   double       c2;
{
   C1 = c1;
   C2 = c2;
}


/*********************************************************************
FUNCTION:  get_sonar_config()
PARAMETERS:
PURPOSE:
```

```
RETURNS:
CALLED BY:
CALLS:    NONE
COMMENTS:  11 September 92 - Dave MacPherson
*****************************************************************/
CONFIGURATION
get_sonar_config(seg_count)
   int          seg_count;
{
   CONFIGURATION  Qsonar;

   Qsonar.x = segment_data_log[0].array[seg_count].tailx;
   Qsonar.y = segment_data_log[0].array[seg_count].taily;
   Qsonar.t = atan2(segment_data_log[0].array[seg_count].heady -
                    segment_data_log[0].array[seg_count].taily,
                    segment_data_log[0].array[seg_count].headx -
                    segment_data_log[0].array[seg_count].tailx);
   Qsonar.k = 0.0;
   return Qsonar;
}
```

```
/*
 Author     : Patrick Byrne
 Date       : 22 Feburary 1994
 File       : sonarlog.h
 Description : Provides extern declarations for functions in sonarLog.c
*/


extern void enable_data_logging();
extern void disable_data_logging();
extern log_data();
extern void set_log_interval();

/*
 * Author     : Patrick Byrne
 * Date       : 22 February 1994
 * File       :sonarlog.c
 * Description : Provides the following Sonar Logging
 * functions:
 *
 * void enable_data_logging(); void disable_data_logging(); void log_data();
 * void set_log_interval();
 *
 */

#include "mml.h"


/
**********************************************************************
 * Procedure: enable_data_logging(n,filetype,filenumber)
 * Description: causes the background system to log data for sonar (n)
 * to a file (filenumber). The data to be logged is specified by an
 * integer flag (filetype). A value of 0 for filetype will cause raw
 * sonar data to be saved, 1 will save global x and y, and 2 will  save
 * line segments. The filenumber may range between 0 and 3 for each of
 * the three types, providing up to 12 data files. Example:
 * enable_data_logging(4,0,0); will cause raw data from sonar #4 to be
 * saved to file 0, while: enable_data_logging(7,2,0);
 * will cause segments for sonar #7 to be saved to file 0. Function
 * increments the service_flag.
 **********************************************************************/
```

```
void
enable_data_logging(n, filetype, filenumber)
   int          n, filetype, filenumber;
{
   global_data_log[filenumber].next = 0;
   if (filetype == 1)
            sonar_table[n].global = 1;
   sonar_table[n].filetype[filetype] = 1;
   sonar_table[n].filenumber[filetype] = filenumber;
   ++service_flag;
}



/*********************************************************************
 * Procedure: disable_data_logging(n,filetype)
 * Description: causes the background system to cease logging data of a
 * given filetype for
 * a sonar n. Decrements the service_flag.
 *********************************************************************/

void
disable_data_logging(n, filetype)
   int          n, filetype;
{
   if ((filetype == 1) && (sonar_table[n].fitting == 0))
            sonar_table[n].global = 0;
   sonar_table[n].filetype[filetype] = 0;
   --service_flag;
}


/*********************************************************************
 * Procedure: log_data(n, type, filenumber,i)
 * Description: this
 * procedure causes data to be written to a file. The filenumber
 * designates which "column" (0,1,2, or 3) of a two dimensional array for
 * that type of data is used. The data array and a counter for each column
 * forms the data structure for each type. The value of i is used to index
 * the seg_list array for storing line segments.
 *********************************************************************/

log_data(n, filetype, filenumber, i)
```

```
    int         n, filetype, filenumber, i;
{
    int         count, interval, next;

    switch (filetype) {
    case 1:
            count = raw_data_log[filenumber].count;
            interval = sonar_table[n].interval;
            if ((count < MAXRAW) && !(count % interval)) {
                    next = raw_data_log[filenumber].next;
                    raw_data_log[filenumber].darray[next] = sonar_table[n].d;
                    raw_data_log[filenumber].xarray[next] = sonar_table[n].x;
                    raw_data_log[filenumber].yarray[next] = sonar_table[n].y;
                    raw_data_log[filenumber].tarray[next] = sonar_table[n].t;
                    raw_data_log[filenumber].next += 1;
            }
            raw_data_log[filenumber].count += 1;
            break;
    case 2:
            if (sonar_table[n].gx == INFINITY){
                    next = global_data_log[filenumber].next;
                    if (global_data_log[filenumber].xarray[next-1] < 9999){
                            count = global_data_log[filenumber].count;
                            interval = sonar_table[n].interval;
                            if ((count < MAXGLOBAL) && !(count % interval)) {
                                    next = global_data_log[filenumber].next;
                                    global_data_log[filenumber].xarray[next]=
INFINITY;
                                    global_data_log[filenumber].yarray[next]=
INFINITY;
                                    global_data_log[filenumber].next += 1;
                            }
                            global_data_log[filenumber].count += 1;
                    }
            }
            else{
                    count = global_data_log[filenumber].count;
                    interval = sonar_table[n].interval;
                    if ((count < MAXGLOBAL) && !(count % interval)) {
                            next = global_data_log[filenumber].next;
                            global_data_log[filenumber].xarray[next]=
sonar_table[n].gx;
```

```
                                global_data_log[filenumber].yarray[next]=
sonar_table[n].gy;
                                global_data_log[filenumber].next += 1;
                        }
                        global_data_log[filenumber].count += 1;
                }
                break;
        case 3:
                count = segment_data_log[filenumber].count;
                if (count < MAXSEGMENT) {
                        segment_data_log[filenumber].array[count] = seg_list[n][i];
    #ifdef SIM
                        printf("\n\nLogging segment data count => %d sonar => %d ",
                            count, n);
                        printf("\nThe Line segment is:
%s%5.1f%s%5.1f%s%5.1f%s%5.1f\n%s%5.1f%s%5.1f\n",
                              " headx = ",
                           segment_data_log[filenumber].array[count].headx,
                              " heady = ",
                           segment_data_log[filenumber].array[count].heady,
                              "  tailx = ",
                           segment_data_log[filenumber].array[count].tailx,
                              "  taily = ",
                           segment_data_log[filenumber].array[count].taily,
                              "  length = ",
                          segment_data_log[filenumber].array[count].length/*
    ,
                              "  Phi  ",
                           segment_data_log[filenumber].array[count].phi*/);
    #endif

        }
                segment_data_log[filenumber].count += 1;
                break;


        }
    }

    /*********************************************************************
     *  Procedure: set_log_interval(n,d)
     * Description: this procedure
     * allows the user to set how often the sonar system writes data to
```

```
 * the raw data or global data files. The interval d is stored at
 * sonar_table[n], and one data point will be recorded for every d data
 * points sensed by the sonar. Default value for interval d is 13, which for
 * a speed of 30 cm/sec and sonar sampling time of 25 msec should
 * record a data point every 10 cm.
**********************************************************************/

void
set_log_interval(n, d)
    int         n, d;
{
    sonar_table[n].interval = d;
}
```

B. USER FILES

```
#include "mml.h"
user()
{
/* File for translation scanning of sonar 4 */
/* Uses logging functions for local trace,  */
/* segment sonar data, and global sonar data*/
/* Pat Byrne 11Nov93                        */
/* Case 1*/

 CONFIGURATION  first, second;
 double s;
 def_configuration(1500.0, 146.0, PI, 0.0, &first);
 def_configuration(800.0, 0.0, HPI, 0.0, &second);
 s = 20.0;
 speed(15.0);

  buffer_loc = index_loc = malloc(300000) ;
  bufloc = indxloc = (double *) malloc(60000);
  loc_tron(2, 0x3f, 30);
  set_rob(&first);
  enable_sonar(RIGHTF);
  size_const(s);
  set_log_interval(RIGHTF, 1);
  enable_linear_fitting(RIGHTF);
  enable_data_logging(RIGHTF, 1, 0);
  enable_data_logging(RIGHTF, 2, 0);
  enable_display_status();
  line(&first);
  bline(&second);
  while(vehicle.x > 1300.0);
  disable_sonar(RIGHTF);
  disable_linear_fitting(RIGHTF);
  disable_data_logging(RIGHTF, 2);
  disable_data_logging(RIGHTF, 1);
  loc_troff();
  motor_on = NO;
  xfer_global_to_host(0, "global7.test");
  xfer_segment_to_host(0, "segment7.test");
  loc_trdump("loc_dump.test");

}
```

```
#include "mml.h"
user()
{
/* File for translation scanning of sonar 7 */
/* Uses logging functions for local trace,  */
/* segment sonar data, and global sonar data*/
/* Pat Byrne 31 Jan 93                      */
/* Case 2 *?
 CONFIGURATION  first, second, third,fourth,fifth,sixth;
 double s;
 def_configuration(0.0, 0.0, 0.0, 0.0, &first);
 def_configuration(100.0, 300.0, HPI, 0.0, &second);
 s = 20.0;
 speed(15.0);
  buffer_loc = index_loc = malloc(300000) ;
  bufloc = indxloc = (double *) malloc(60000);
  loc_tron(2, 0x3f, 30);
  set_rob(&first);
  enable_sonar(RIGHTF);
  size_const(s);
  set_log_interval(RIGHTF, 1);
  enable_linear_fitting(RIGHTF);
  enable_data_logging(RIGHTF, 1, 0);
  enable_data_logging(RIGHTF, 2, 0);
  enable_display_status();
  line(&first);
  line(&second);
  while(vehicle.y < 150.0);
  disable_sonar(RIGHTF);
  disable_linear_fitting(RIGHTF);
  disable_data_logging(RIGHTF, 2);
  disable_data_logging(RIGHTF, 1);
  loc_troff();
  motor_on = NO;
  xfer_global_to_host(0, "global7.test");
  xfer_segment_to_host(0, "segment7.test");
  loc_trdump("loc_dump.test");

}
```

```
#include "mml.h"
user()
{
/* File for translation scanning of sonar 7 */
/* Uses logging functions for local trace,  */
/* segment sonar data, and global sonar data*/
/* Pat Byrne 11Nov93                        */
/* Case 3*/
 CONFIGURATION  first, second, third,fourth,fifth,sixth;
 double s;
 def_configuration(951.6, -50.0, HPI, 0.0, &first);
 def_configuration(951.6, -30.0, HPI/2, 0.0, &second);
 s = 20.0;
 speed(15.0);
  buffer_loc = index_loc = malloc(300000) ;
  bufloc = indxloc = (double *) malloc(60000);
  loc_tron(2, 0x3f, 30);
  set_rob(&first);
  enable_sonar(RIGHTF);
  size_const(s);
  set_log_interval(RIGHTF, 1);
  enable_linear_fitting(RIGHTF);
  enable_data_logging(RIGHTF, 1, 0);
  enable_data_logging(RIGHTF, 2, 0);
  enable_display_status();
  line(&first);
  line(&second);
  while(vehicle.x < 1150.0);
  disable_sonar(RIGHTF);
  disable_linear_fitting(RIGHTF);
  disable_data_logging(RIGHTF, 2);
  disable_data_logging(RIGHTF, 1);
  loc_troff();
  motor_on = NO;
  xfer_global_to_host(0, "global7.test");
  xfer_segment_to_host(0, "segment7.test");
  loc_trdump("loc_dump.test");

}
```

```
/* File for rotational scanning of sonar 3 */
/* Pat Byrne Nov30, 93                     */
/* Case 4                                  */
#include "mml.h"
user()
{
 CONFIGURATION  p1;
 void initialize();
 void cleanup();
 def_configuration(1200.0, 146.0, 0.0, 0.0, &p1);
  speed(15.0);
  initialize(p1);
 set_rob(&p1);
 rotate(DPI);
 while(vehicle.t < PI );
 cleanup();
}
void initialize(p1)
 {
  double s = 20.0;
  buffer_loc = index_loc = malloc(300000) ;
  bufloc = indxloc = (double *) malloc(60000);
  loc_tron(2, 0x3f, 30);
  set_rob(&p1);
  enable_sonar(FRONTR);
  speed(15.0);
  size_const(s);
  set_log_interval(FRONTR, 1);
  enable_linear_fitting(FRONTR);
  enable_data_logging(FRONTR, 1, 0);
  enable_data_logging(FRONTR, 2, 0);
 }
void cleanup()
 {
   disable_sonar(FRONTR);
  generate_segment(FRONTR);
  disable_data_logging(FRONTR, 2);
  loc_troff();
  motor_on = NO;
    xfer_global_to_host(0, "global.test");
  xfer_segment_to_host(0, "segment3.test");
  loc_trdump("loc_dump.test");
 }
```

```
/* File for rotational scanning of sonar 3 */
/* Pat Byrne Nov30, 93                    */
#include "mml.h"
/* Case 5   */
user()
{
 CONFIGURATION  p1;
 void initialize();
 void cleanup();
 def_configuration(951.6, -504.5, 0.0, 0.0, &p1);
 speed(15.0);
 initialize(p1);
 set_rob(&p1);
 rotate(DPI);
 while(vehicle.t < PI );
 cleanup();
}
void initialize(p1)
 {
  double s = 20.0;
  buffer_loc = index_loc = malloc(300000) ;
  bufloc = indxloc = (double *) malloc(60000);
  loc_tron(2, 0x3f, 30);
  set_rob(&p1);
  enable_sonar(FRONTR);
  speed(15.0);
  size_const(s);
  set_log_interval(FRONTR, 1);
  enable_linear_fitting(FRONTR);
  enable_data_logging(FRONTR, 1, 0);
  enable_data_logging(FRONTR, 2, 0);
 }
void cleanup()
 {
    disable_sonar(FRONTR);
  generate_segment(FRONTR);
  disable_data_logging(FRONTR, 2);
  loc_troff();
  motor_on = NO;
    xfer_global_to_host(0, "global.test");
  xfer_segment_to_host(0, "segment3.test");
  loc_trdump("loc_dump.test");
 }
```

```
/* File for rotational scanning of sonar 3 */
/* Pat Byrne Nov30, 93                     */
/* Case 6                    */
#include "mml.h"
user()
{
 CONFIGURATION  p1;
 void initialize();
 void cleanup();
 def_configuration(781.0, 100.00, PI, 0.0, &p1);
 speed(15.0);
 initialize(p1);
 set_rob(&p1);
 rotate(PI);
 while(vehicle.t < DPI );
 cleanup();
}
void initialize(p1)
  {
   double s = 20.0;
   buffer_loc = index_loc = malloc(300000) ;
   bufloc = indxloc = (double *) malloc(60000);
   loc_tron(2, 0x3f, 30);
   set_rob(&p1);
   enable_sonar(FRONTR);
   speed(15.0);
   size_const(s);
   set_log_interval(FRONTR, 1);
   enable_linear_fitting(FRONTR);
   enable_data_logging(FRONTR, 1, 0);
   enable_data_logging(FRONTR, 2, 0);
  }
void cleanup()
 {
   disable_sonar(FRONTR);
   generate_segment(FRONTR);
   disable_data_logging(FRONTR, 2);
   loc_troff();
   motor_on = NO;
   xfer_global_to_host(0, "global.test");
   xfer_segment_to_host(0, "segment3.test");
   loc_trdump("loc_dump.test");
  }
```

```
#include "mml.h"

user()
{
/* File for translation scanning of sonar 7 */
/* Uses logging functions for local trace,  */
/* segment sonar data, and global sonar data*/
/* Pat Byrne 1Feb94                         */
/* Case 7                                   */
 CONFIGURATION  first, second;
 double s;
 def_configuration(0.0, 0.0, 0.0, 0.0, &first);
 def_configuration(800.0, 0.0, HPI, 0.0, &second);
 s = 20.0;
 speed(15.0);
 buffer_loc = index_loc = malloc(300000) ;
 bufloc = indxloc = (double *) malloc(60000);
 loc_tron(2, 0x3f, 30);
 set_rob(&first);
 enable_sonar(RIGHTF);
 size_const(s);
 set_log_interval(RIGHTF, 1);
 enable_linear_fitting(RIGHTF);
 enable_data_logging(RIGHTF, 1, 0);
 enable_data_logging(RIGHTF, 2, 0);
 enable_display_status();
 line(&first);
 bline(&second);
 while(vehicle.x  < 300.0);
  disable_sonar(RIGHTF);
  disable_linear_fitting(RIGHTF);
  disable_data_logging(RIGHTF, 2);
  disable_data_logging(RIGHTF, 1);
  loc_troff();
  motor_on = NO;
  xfer_global_to_host(0, "global7.test");
  xfer_segment_to_host(0, "segment7.test");
  loc_trdump("loc_dump.test");
}
```

```
#include "mml.h"
user()
{
/* File for rotational scanning of sonar 3 */
/* Uses logging functions for local trace,  */
/* segment sonar data, and global sonar data*/
/* Pat Byrne 11Nov93                */
/* case 8                  */
 CONFIGURATION  first, second;
 double s;
 def_configuration(1500.0, 146.0, PI, 0.0, &first);
 def_configuration(800.0, 0.0, HPI, 0.0, &second);
 s = 20.0;
 speed(15.0);
  buffer_loc = index_loc = malloc(300000) ;
  bufloc = indxloc = (double *) malloc(60000);
  loc_tron(2, 0x3f, 30);
  set_rob(&first);
  enable_sonar(RIGHTF);
  size_const(s);
  set_log_interval(RIGHTF, 1);
  enable_linear_fitting(RIGHTF);
  enable_data_logging(RIGHTF, 1, 0);
  enable_data_logging(RIGHTF, 2, 0);
 enable_display_status();
 line(&first);
 bline(&second);
 while(vehicle.x > 1300.0);
  disable_sonar(RIGHTF);
  disable_linear_fitting(RIGHTF);
  disable_data_logging(RIGHTF, 2);
  disable_data_logging(RIGHTF, 1);
  loc_troff();
  motor_on = NO;
  xfer_global_to_host(0, "global7.test");
  xfer_segment_to_host(0, "segment7.test");
  loc_trdump("loc_dump.test");

}
```

```
#include "mml.h"
/* obstacle example one  */
user()
{
 double hit11;
 CONFIGURATION  p1, p3, posit1;
 def_configuration(1051.0, 146.0, 0.0, 0.0, &p1);
 def_configuration(1651.0,46.0,0.0,0.0, &p3);
 speed(15.0);
 buffer_loc = index_loc = malloc(300000) ;
 bufloc = indxloc = (double *) malloc(60000);
 loc_tron(2, 0x3f, 30);

 set_rob(&p1);
 enable_sonar(FRONTR);
 set_log_interval(FRONTR, 1);
 enable_data_logging(FRONTR, 1, 0);
 hit11 = sonar(FRONTR);
  line(&p1);
    while(hit11 >= 100.0 || hit11 == 0.0 ){
            hit11 = sonar(FRONTR);
    }
 skip();
 line(&p3);
 get_rob0(&posit1);
 while(posit1.x < 1351){
 get_rob0(&posit1);
 }
 skip();
 line(&p1);

 get_rob0(&posit1);
 while (posit1.x < 1651){
 get_rob0(&posit1);
 }
 disable_sonar(FRONTR);
 disable_data_logging(FRONTR, 1);
 loc_troff();
 stop0();
 motor_on = OFF;
 xfer_global_to_host(0, "global7.AVOID");
 loc_trdump("loc_dump.AVOID");
}
```

```
#include "mml.h"
/* example of obstacle avoidance 2 */
user()
{
    double      hit11;
    CONFIGURATION   p1, p3, p4, start, posit1;
    def_configuration(0.0,0.0,0.0,0.0, &start);
    def_configuration(600.0, 0.0, 0.0, 0.0, &p1);
    def_configuration(0.0, -100.0, 0.0, 0.0, &p3);
    speed(15.0);
    buffer_loc = index_loc = malloc(300000);
    bufloc = indxloc = (double *) malloc(60000);
    loc_tron(2, 0x3f, 30);
    set_rob(&start);

    enable_sonar(FRONTR);
    enable_sonar(LEFTF);
    hit11 = sonar(FRONTR);

    line(&p1);
    hit11 = 9999.0;
    while (hit11 >= 100.0 || hit11 == 0.0) {
            hit11 = sonar(FRONTR);
    }
    skip();
    line(&p3);
    hit11 = sonar(LEFTF);
    while (hit11 >= 100.0 || hit11 == 0.0){
            hit11 = sonar(LEFTF);
    }
    while (hit11 <= 100.0 || hit11 == 0.0){
            hit11 = sonar(LEFTF);
    }

    skip();
    line(&p1);
    get_rob0(&posit1);
    while(posit1.x < 600.0){
            get_rob0(&posit1);
    }

    disable_sonar(FRONTR);
    disable_sonar(LEFTF);
```

```
    loc_troff();
    stop0();
    motor_on = OFF;
    loc_trdump("loc_dump.AVOID");
}
```

LIST OF REFERENCES

1. Sherfey, S., *A Mobile Robot Sonar System,* Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1991.

2 Kanayama, Y., Noguchi, T., "Spatial Learning by an Autonomous Mobile Robot with Ultrasonic Sensors", University of California Department of Computer Science *Technical Report* TRCS89-06, February, 1989.

3 Kanayama, Y., "Mathematical Theory of Robotics: Introduction to 2D Spatial Reasoning", *Lecture Notes of the Advanced Robotics Course*, Department of Computer Science, Naval Postgraduate School, Winter Quarter 1994.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943-5101	2
3.	Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
4.	Dr. Yutaka Kanayama, Code CS/KA Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
5.	Dr. Man-Tak Shing, CS/SH Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
6.	Lt. Patrick G. Byrne 569 Christopher Lane Doylestown, PA 18901	1